BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Patent Application for

METHODS AND APPARATUS FOR REMOTE PROCESS CONTROL

Appendix II

(Object Manager API Specification)

I/A Series

B0193BC
Rev E

Object Manager Calls

October 31, 1995

FOXBORO®
A SIEBE COMPANY

SOFTWARE LICENSE AND COPYRIGHT INFORMATION

Before using the Foxboro supplied software supported by this Foxboro documentation, you should read and understand the following information concerning copyrighted software.

1. The license provisions in the Foxboro Software License for your system govern your obligations and usage rights to the software described in this documentation. If any portion of those license provisions is violated, The Foxboro Company will no longer provide you with support services and assumes no further responsibilities for your system or its operation.
2. All software issued by The Foxboro Company, and copies of the software that you are specifically permitted to make, are protected in accordance with Federal copyright laws. It is illegal to make copies of any software media provided to you by The Foxboro Company for any purpose other than those purposes mentioned in the Foxboro Software License.

Contents

THIS PAGE BLANK (USPTO)

Preface

This document is for process control software engineers who wish to write application programs to run on the I/A Series System.

Readers are assumed to know the C or FORTRAN programming languages. They are also assumed to be familiar with the VENIX (or UNIX) operating system.

I/A Series reference Documents:

SunOS Reference Manual I, II, III (B0193LC)
VENIX User Reference Manual (B0193BV)
VENIX Programmer's Reference Manual (B0193BX)
VENIX Administrator's Reference Manual (B0193BW)
VENIX Support Tools Guide (B0193CA)
50 Series Program Development (B0193LQ)
Program Development VENIX (B0193BA)
Inter-Process Communications Calls (B0193BB)
Human Interface Calls (B0193BD)
Miscellaneous Calls (B0193BE)
Supervisory Setpoint Control (B0193RY)
System Messages (B0193CG)

Revision Information

The following changes were made for Release 4.2:

All calls in each Section were listed alphabetically. Added an introduction to each Section which lists the calls in the Section.

Section 1. Object Manager Concepts

- Added CP40 to Object Manager Parameter Table.

Section 2. C Calls to Get/Set Object Values

- Added the following new calls:
- st_setval, st_setcnf, st_om_setval, st_omset_confirm, st_set_list and st_setlist_confirm.
- Modified the data type and return code descriptions for the setval, set_confirm, om_setval, om_set_confirm, setval_list and set_confirm_list.
- Changed call set_confirm_list to set_cnf_list.

- Deleted return codes: NOT_SETTABLE, SECURED, TO_REAL_CONV_ERROR, TO_INTEGER_CONV_ERROR, TO_STRING_CONV_ERROR, TO_BOOL_CONV_ERROR, from serval, set_confirm, om_serval, om_set_confirm, serval_list and set_cnf_list.

Section 3. C Calls to Access Update Sets of Variable

- Added the new call: omwrstat.
- Changed the value structure descriptions for "index" and "value" in the omwrite call.
- Added EWRITERROR return codes and changed return code descriptions for omwrite call.

Section 5. FORTRAN Function to Get/Set Object Values

- Added command strong to compile FORTRAN programs.

Appendix B. OM Error Codes

- Added OM Error Codes, Appendix B.

Appendix C. OM Calls

- Added OM C Calls
- Added OM FORTRAN Calls

1. Object Manager Concepts

An object is a construct that the Object Manager can find by name, relieving you of the need to know its location or path name. The operating system automatically declares process variables, letterbug names, and device logical names to be objects. Object Manager calls let you create and delete object names and read and write (get and set) object values.

Some of the Object Manager calls themselves invoke calls from other subsystems. When this happens, some return codes might actually come from the other subsystem. The exact code is passed back to the calling task whenever possible, and the explanations are included with each call.

The Object Manager supports two kinds of objects: shared objects and process-control objects. Their characteristics are given below.

Shared Objects Variables	Control Objects or Process
14 character name (maximum)	32 character name
You can create, delete, read and write shared objects	You can only read and write control objects
Can be of all object types	Can only be of object type VARIABLE

1.1 Object and Data Types

An object can have one of the following types:

- VARIABLE — A shared and process (control compound) variable
- ALIAS — An object whose value is the name of another object
- DEVICE/ LETTERBUG — The letterbug id or logical name of an external device like a station or printer
- PROCESS — A program in execution; a task

The regular shared VARIABLE object type can have any of the following data types:

- CHARACTER
- INTEGER
- FLOAT
- STRING
- OM_BOOL (character, values = true, false)
- OM_LNG_INT (long integer)
- OM_S_PKBOL (packed boolean)
- OM_L_PKBOL (long packed boolean)

An ALIAS object type can only have STRING data type.

The object types DEVICE and PROCESS are typically, but not always, used with global_find to get the address of the station where the device or task is resident. Since the object types DEVICE/LETTERBUG and PROCESS do not have values, they cannot have any data types.

The Object Manager does not restrict the data type of control objects (VARIABLE object type); they can be of any data type allowed by the control software.

Integers are signed, unless the block parameter definition specifies otherwise. Reals adhere to the IEEE standard.

The data types are stored in the status word of each object's value record structure.

You have to make sure that the ipc.h include file is available, as the Object Manager include file "includes" it as well.

1.2 Object Names

The maximum length for shared object names is 14 characters. Only alphanumerics and the underscore are permitted for types variable and alias. Types device and process can use any character. Duplicate object names are permitted only if they are of different object types.

Control objects (process variables) and device names are exceptions. The Object Manager does not restrict them, although their respective configurators do. Process variables can have names up to 32 characters to accommodate the compound, block, and parameter names.

Refer to the *Integrated Control Configurator* and *System Configurator* documents for more information on these names.

The complete OM name used to get ECB configuration and measurement parameters has the following format:

`<letterbug>_ECB:<FBM_ID>.<I/A_name>`

where letterbug is replaced with the letterbug of the Tank Processor, FBM_ID is replaced with the HIU letterbug, and I/A_name is the parameter name.

Example: To get the pressure transmitter 1 frequency of a HIU with the letterbug of HTG131, which is connected to a Tank Processor with the letterbug of TP1013, the complete OM variable name is TP1013_ECB:HTG131.C105.

All parameters that begin with BYT are character arrays. All parameters that begin with NAME are character arrays. All other parameters are floating point numbers. Refer to the control documents for parameter names. Names in OM open list should be uppercase.

1.3 The Import List and the Object Directory

The import list contains the names of objects from other stations. Listed with each name is an index to its address in the address table. This list lets the Object Manager find objects without sending out a broadcast to locate objects in remote stations. Objects are added to the list as the result of user requests.

If you are not going to get an object many times in a program, you are better off not bothering to import it. There is only one import list for the station, and all tasks have to share it, so for efficiency, you should unimport all objects when you exit.

The get/set calls can import objects as specified. The object directory contains all the shared objects created by tasks in the local station. They automatically become globally known when you create them. The object directory contains pointers to the objects' values, if any.

NOTE: You must close all OM lists before any exit from the program occurs.

1.4 Variable Scan Rates

The Object Manager lets you specify a scan rate for change-driven lists. This variable scan rate is the rate at which the station that is the source of the data sends updates to the station that opened the list. If you have programs written before variable scan rates were available, you should change your code to select the appropriate scan rate.

You must specify the scan rate for an OMOPEN list in the om_header_node structure in a C program. It is in intervals of 1/2 second. A value of one causes a scan rate of 1/2 second, a value of four a scan rate of 2 seconds, and so on.

The range of valid scan rates is from 1/2 second to 120 seconds. An invalid scan rate is forced to the default value of 1/2 second.

The following example is user code setting the scan rate to 10 seconds before performing the *OMOPEN* request.

```
#define SCAN_CYCLES  20
struct om_header_node     *hdr_ptr;

hdr_ptr = (struct om_header_node*)
          malloc ( size of struct om_header_node);
if ( hdr_ptr != (struct om_header_node*) NULL ).
{
    /* Set up for omopen. */
    hdr_ptr->scan_rate =  SCAN_CYCLES;

    /*  Perform OMOPEN. */
 }
 else
```

The FORTRAN format is as specified later for *OMOPEN*.

1.5 Object Manager Parameter Table

This table defines set and maximum values for certain Object Manager parameters by station. A brief explanation of each item in the table follows the table.

Description	50s	AP10/20	PWs	WPs	CPs	CMs
Max # of OMOPEN Lists	100#	50	50	50	(60/360)	50
Max # of Points per List	255#	255	255	255	255	255
Max # of Shared Objects	750#	100/750	750	60	100	100
Max # of Imported Objects	100#	50	50	50	50	50
Max # of Scanner Entries	100#	50	50	0	50/1600**	50
GETVAL Timeout (seconds)	12	12	12	12	150/12	12
SET_CONFIRM Timeout (sec)	12	12	12	12	12	12
Max CHANGE Queues	100#	50	50	5	5/10***	5
Max # of IPC Connections	30#	12	12	12/30*	12/30*	12

*12 for WP20 and CP10, 30 for WP30 and CP30/CP40.
**50 for CP10, 150 for CP30, 600 for CP40.
***5 for CP10, 10 for CP30/CP40.
#V3.1 release on 50 Series maximums are configurable

Some items are subject to constraints from other sub-systems (the number of connections or the number of points scanned, for example)..

OMOPEN Lists — The maximum number of change-driven omopen lists.

Points per List — The maximum number of change-driven points for one omopen list.

Shared Objects — The maximum number of shared objects that can exist in a station at one time.

Imported Objects — The maximum number of objects that can be in an import list at one time.

Scanner Entries — The number of scanner entries that one station can scan. Each entry corresponds to (part of) a list and can contain up to 20 points. The actual number of points a station can scan depends on the number of omopen lists and the size of each list. How many scanner entries are used for an omopen list is computed as follows:

entries = (# points in station for list +19)/20

50 Series stations are an exception and contain up to 255 points per entry and therefore have a one to one relationship between a list and scanner entry.

Example:

A WP20 opens a list for 86 points that has 40 points in CP1, 45 points in CP2, and 1 point in CP3.

CP1 scanner entries = (40+19)/20 = 2

CP2 scanner entries = (45+19)/20 = 3

CP3 scanner entries = (1+19)/20 = 1

For 50 Series = 1 entry per list

GETVAL Timeout

The number of seconds the Object Manager waits to receive data on a GETVAL request. If the object does not exist, the requester suspends for 12 seconds.

SET_CONFIRM Timeout

The number of seconds the Object Manager waits to get a reply to a SET_CONFIRM request. If the station with the object is down, the requester suspends for 12 seconds.

CHANGE Queues

The number of queues available for open lists that want notification of changes. These queues are based on the process id in a particular station.

1.6 Object Manager Databases

The databases of the Object Manager for 50 Series reside in shared memory. This memory is divided into three partitions. The first partition is static and contains the static tables. The second partition is a valloc pool for open lists. The third partition is another valloc pool dedicated to strings created for shared objects (see Figure 1-1 and Figure 1-2).

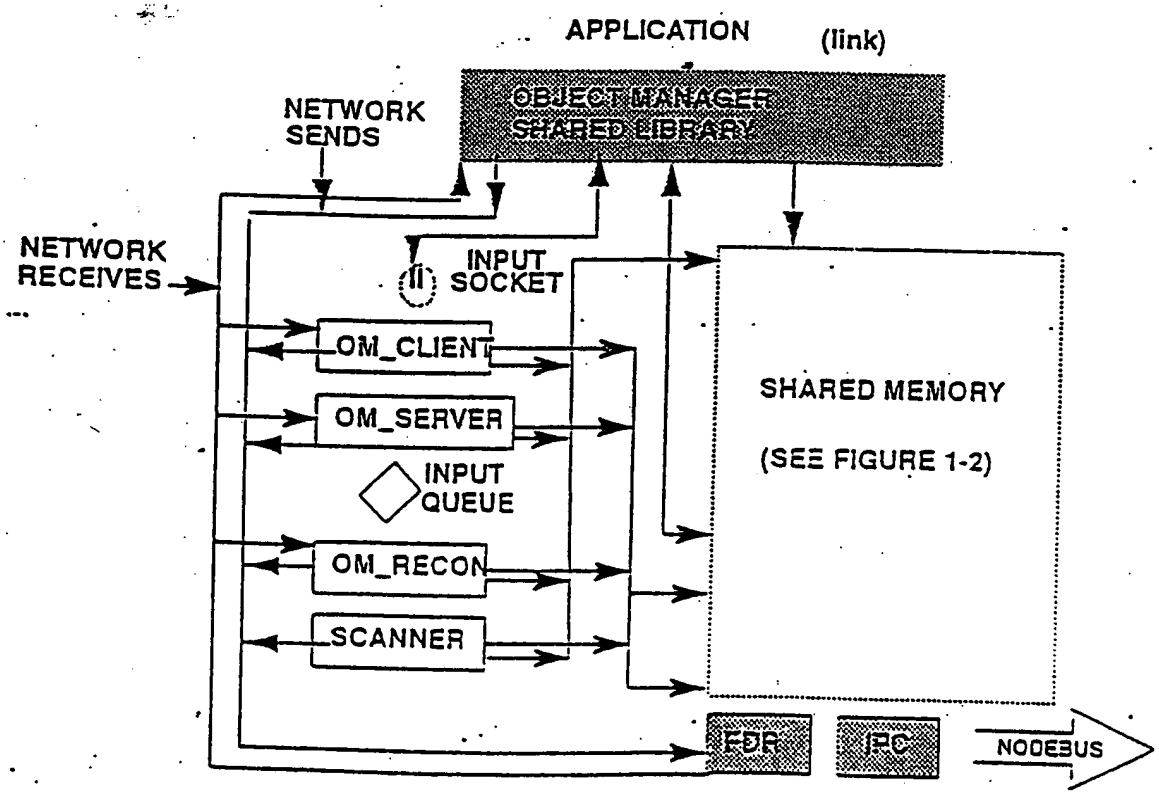


Figure 1-1. Object Manager Software Components

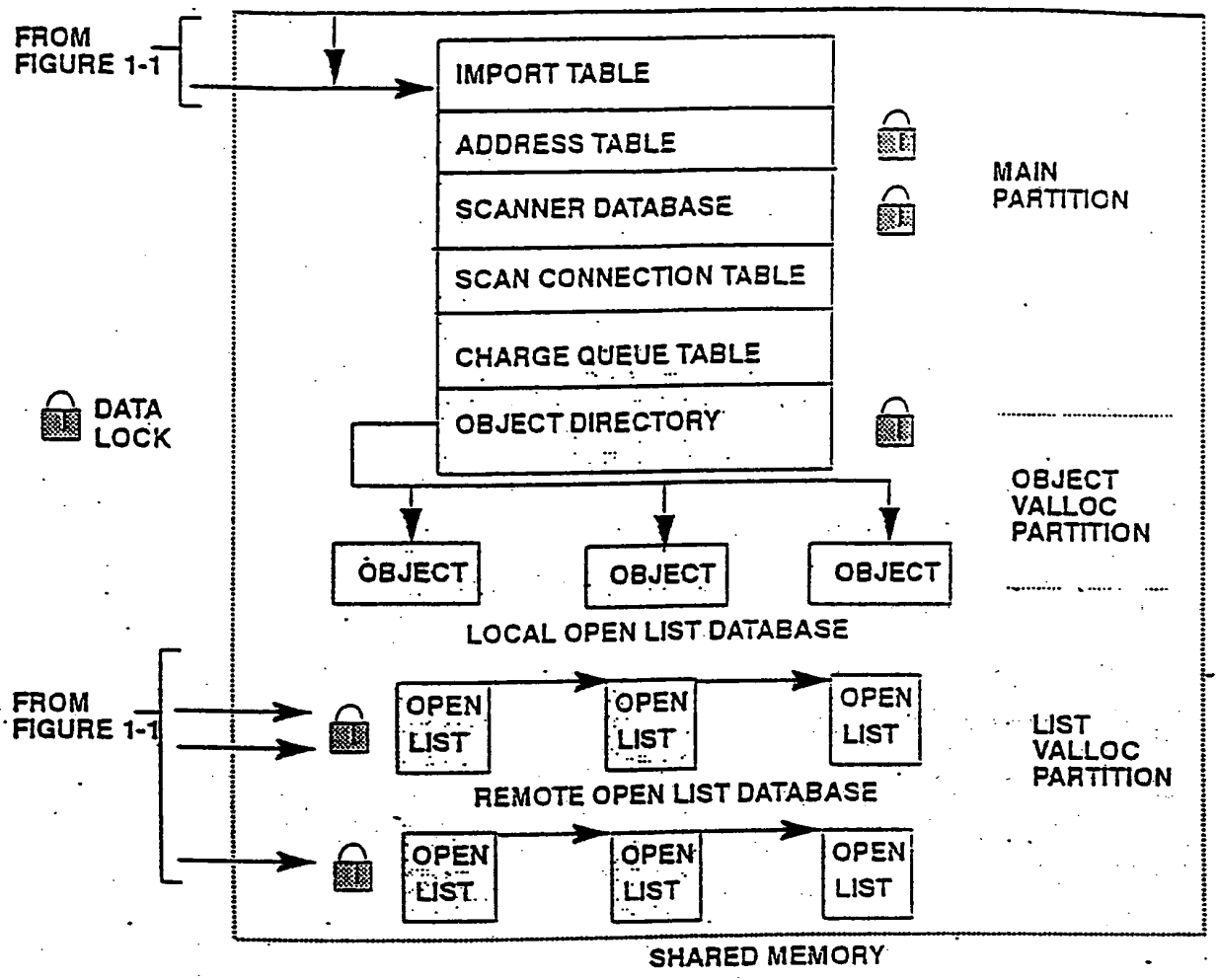


Figure 1-2. Object Manager Software Components (Cont.)

1.6.1 Open Points Database

The open points database is a linked list of descriptors (or header nodes) which describes lists of objects opened in this station. This database is further split into two portions: Remote Lists – those opened from remote stations, and Local Lists – those opened by applications locally. Remote Lists do not have a Network Address Table.

Separate locks (read and write) are maintained for each of the two portions. Since the Object Manager server operates (and locks) primarily on the Remote portion of the database, it does not usually interfere with user processes which access primarily the Local portion. As mentioned above, the lists are valloced from one distinct partition.

1.6.2 Scanner Database

The scanner database is an indicator of which objects in an open list are to be scanned. All Remote Lists that are not opened for *write only* access, are scanned. Objects in Local Lists are

scanned only if they are local (created in this station). Single connect control variables are only scanned once. Deleted objects in lists are skipped, after notification has been sent.

The number of variable record pointers that can be contained in a row of the database (which corresponds to one list exactly), is equal to the maximum number of objects in a list (currently 255).

list ptr	open var ptr	open var ptr	.....	open var ptr

1.6.3 Object Directory

The object directory contains the name of the shared object, the export status, and the pointer to the dynamically allocated value record (as described above, from a separate shared memory partition). Process names and device names have null value_rec pointers. A read lock and a write lock synchronizes access.

entry count		
name [ ]	export status	value record ptr

1.6.4 Import Table

The import table stores variable names with indices to the Object Manager network address table. A read lock and a write lock synchronizes access.

entry count	
name [ ]	network address index

1.6.5 Object Manager Network Address Table

This table is an array of network addresses pointed to by the import list and the object directory value records. A read lock and a write lock synchronizes access.

PSAP	user count

1.6.6 Scanner Connection Table

The scanner connection table contains an array of network addresses for the connected stations and the number of communication retrys attempted for each connection. The connect status flag indicates whether the connection needs to be disconnected, and whether the entry is needed for the creation of a new connection.

PSAP index	channel id	reference count	retry count	status

1.6.7 Server Connection Table

The server connection table contains the channel id and the Primary System Address Protocol (PSAP address) of all remote stations.

channel id	PSAP

2. C Calls to Get/Set Object Values

These calls are for getting (reading) or setting (writing) object values. As connectionless calls they are best suited for situations in which you only want a single transfer of data.

As explained in the document *Inter-Process Communications Calls*, connectionless calls are calls that transfer data between tasks without having established an IPC connection (channel) first.

The Object Manager registers your task with IPC for connectionless service. You do not have to worry about it.

The OM C calls to get/set object values are listed in this Section in alphabetical order and summarized in Table 2-1.

Table 2-1. C calls to Get/Set Object Values

Paragraph No.	Call	Function
2.1	getval	Get the value of an object
2.2	getval_list	Get the value of an object
2.3	om_getval	Get the value of an object
2.4	om_set_confirm	Set the value of an object
2.5	om_setval	Set the value of an object
2.6	set_confirm	Set the value of an object
2.7	set_cnf_list	Set the value of an object
2.8	setval	Set the value of an object
2.9	setval_list	Set the value of an object
2.10	st_omset_confirm	Set the value and/or status of an object
2.11	st_om_setval	Set the value and/or status of an object
2.12	st_setcnf	Set the value and/or status of an object
2.13	st_setlist_confirm	Set the value and/or status of an object
2.14	st_set_list	Set the value and/or status of an object
2.15	st_setval	Set the value and/or status of an object

2.1 getval – Get the Value of an Object

getval gets the value of the specified object. An object is identified by its name and type. The object name can be either a shared object name or a compound name. If the object name is a compound name, then the object type must be variable. If the object name is a shared object name, then object types can be either variable or alias. Object type shared variables can be data types character, integer, long integer, float, string, boolean, packed boolean, and long packed boolean. Object type alias has a data type string. There is no data value record for object types process and device. *getval* is synchronous; it suspends your task until the transfer is complete.

Format

```
int getval(<name>,<obj_type>,<import>,<value>,<status>,data_len>);
    char            *name, *value;
    int             obj_type, import, *data_len;
    unsigned int    *status;
```

Where:

`*name` —	A character pointer to the name of the object.
`obj_type`	The named object's type; VARIABLE or ALIAS.
`import`	1 = place the object's name on the import list. 0 = do not place it on the import list.
`*value`	A pointer to the location where the Object Manager is to put the object's value. Specify its size in bytes using `data_len`.
`*status`	A pointer to the location where the Object Manager is to return the object's status (data type). An ALIAS object type can only have STRING data type. The regular shared VARIABLE object type can have a data type of CHARACTER, INTEGER, FLOAT, STRING, OM_BOOL, OM_LNG_INT (long integer), OM_S_PKBOL (packed boolean), or OM_L_PKBOL (long packed boolean). The Object Manager does not restrict process VARIABLE data types, but they can be of types CHARACTER, INTEGER, REAL, STRING, OM_BOOL (boolean), and LONG_INT and POINTER (both long integer). Integers are signed, unless the block parameter definition specifies otherwise. Reals adhere to the IEEE standard.
`*data_len`	A pointer to the length (bytes) allocated for `value`. OM returns actual length.

Return Codes:

`OM_SUCCESS`	No errors.
`ENOTFOUND`	Object not found (locally or globally).

EBADTYPE	Illegal object type.
ENOSPACE	No memory available; no value returned.
EBADNAME	Invalid object name.
EBADLEN	Shared variable is too big for data_len.
ENOVALUE	Specified object type has no "value".
EBADVREC	"Value" is corrupt, invalid data type.
EIPCRET	You have an unspecified IPC error.
TO_BIG	Process variable is too big for data_len.

Call Notes:

1. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects *getval* ignores that option. However, the value is still returned.
2. If you get an error return, the Object Manager may still return a value, but it is garbage. Do *not* send the returned value to useful data areas unless there is no error return.
3. Return code ENOTFOUND might mean that the object does not exist locally or globally. If the object exists in a remote station, then that station might be down. In either case, The Object Manager removes the object's name from the import list.
4. If you do not make data_len large enough, value will be too small to hold the returned value. In such a case getval returns as much data in value as will fit.
5. If data_len is too large, value will contain unused bytes. *getval* does not close-up this unused memory space.
6. Status, data_len, and value are modified regardless of error return.

2.2 getval_list – Get the Value of an Object

getval_list gets the value of the specified object. This call is a variation of the *om_getval* call. It differs only in the way the PSAP is specified.

Instead of the `psap_ptr` input argument used in *om_getval*(), the *getval_list*() call uses an <open_id> and an <ov_index> argument. The < open_id> argument specifies the ID of an open variables list. The list need not be optimized, but it must currently be opened. The <ov_index> specifies the index of a variable entry within the list. The variable must be connected. It is assumed that the variable in the list and the target object reside in the same station. (The variable and the object may or may not be one in the same.) The PSAP associated with that variable identifies the station to which the getval message will be sent. The message will not be broadcast.

getval_list is synchronous; it suspends your task until the transfer is complete.

Format:

```
int  getval_list(<name>,<obj_type>,<import>,<value>,<status>,<data_len>,
<open_id>,<ov_index>)
    char            *name, *value;
    int             obj_type, import, *data_len, open_id,ov_index;
    unsigned int    *status;
```

Where:

`*name`	A character pointer to the name of the object.
`obj_type`	The named object's type; VARIABLE or ALIAS.
`import`	1 = place the object's name on the import list. 0 = do not place it on the import list.
`*value`	A pointer to the location where the Object Manager is to put the object's value. Specify its size in bytes using data_len.
`*status`	A pointer to the location where the Object Manager is to return the object's status (data type). An ALIAS object type can only have STRING data type. The regular shared VARIABLE object type can have a data type of CHARACTER, INTEGER, FLOAT, STRING, OM_BOOL, OM_LNG_INT (long integer), OM_S_PKBOL (packed boolean), or OM_L_PKBOL (long packed boolean). The Object Manager does not restrict process VARIABLE data types, but they can be of types CHARACTER, INTEGER, REAL, STRING, BOOL (boolean), and LONG_INT and POINTER (both long integer). Integers are signed, unless the block parameter definition specifies otherwise. Reals adhere to the IEEE standard.
`*data_len`	A pointer to the length (bytes) allocated for "value." OM returns actual length.
`open_id`	ID of an open variables list.

ov_index	Index of a variable entry within the list.

Return Codes:

OM_SUCCESS	No errors.
ENOTFOUND	The name to get was found in the import list, but IPC returned an error which indicates the message was rejected because station may be down. The object name is removed from the import list.
EBADTYPE	Illegal object type argument.
ENOSPACE	No memory available; no value returned.
EBADNAME	Invalid object name.
EBADLEN	Shared variable is too big for data_len.
ENOVALUE	Specified object type has no "value" The caller requested to get the value of an object type process or device.
EBADVREC	"Value" record is corrupt, invalid data type.
EIMPFULL	Import table full; value is returned.
EBADINDEX	The specified ov_index is out of range, or selects an entry which is not currently in use.
ENOTOPENED	The specified open_id does not correspond to an open variables list.
ENOTACTIVE	Not registered with IPC for connectionless service.
EIPCRET	You have an unspecified IPC error.

Call Notes:

1. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects *getval_list* ignores that option. However, the value is still returned.
2. If you get an error return, the Object Manager may still return a value, but it is garbage. Do NOT send the returned value to useful data areas unless there is no error return).
3. If you do not make data_len large enough, value will be too small to hold the returned value. In such a case getval returns as much data in value as will fit.
4. If data_len is too large, value will contain unused bytes. *getval_list* does not close-up this unused memory space.
5. Status, data_len, and value are modified regardless of error return.

2.3 om_getval – Get the Value of an Object

om_getval gets the value of a specified object. This call is a functional superset of the *getval*() call. It uses Primary System Address Protocol (PSAP) pointer as an additional argument. If the pointer value is NULL, *om_getval*() behaves as a *getval*() call, and the *getval* message is broadcast. *om_getval* is synchronous; it suspends your task until the transfer is complete.

Format:

```
int  om_getval(<name>,<obj_type>,<import>,<value>,<status>,<data_len>,
<psap_ptr>)

    char            *name, *value;
    int             obj_type, import, *data_len;
    unsigned int    *status;
    PSAP-ADDR       *psap_ptr;
```

Where:

`*name`	A character pointer to the name of the object.
`obj_type`	The named object's type; VARIABLE, ALIAS.
`import`	1 = place the object's name on the import list. 0 = do not place it on the import list.
`*value`	A pointer to the location where the Object Manager is to put the object's value. Specify its size in bytes using `data_len`.
`*status`	A pointer to the location where the Object Manager is to return the object's status (data type). An ALIAS object type can only have STRING data type. The regular shared VARIABLE object type can have a data type of CHARACTER, INTEGER, FLOAT, STRING, OM_BOOL, OM_LNG_INT (long integer), OM_S_PKBOL, n or OM_L_PKBOL. The Object Manager does not restrict process VARIABLE data types, but they can be of types CHARACTER, INTEGER, REAL, STRING, BOOL (boolean), and LONG_INT and POINTER (both long integer), OM_S_PKBOL (packed boolean), and OM_L_PKBOL (long packed boolean). Integers are signed, unless the block parameter definition specifies otherwise. Reals adhere to the IEEE standard.
`*data_len`	A pointer to the length (bytes) allocated for `value`. OM returns actual length.
`psap_ptr`	Pointer to the PSAP identifying the station on which the variable is found. A value of (PSAP_ADDR *)NULL indicates that no PSAP has been provided, so the station must be found using a broadcast message.

Return Codes:

`OM_SUCCESS`	No errors.

ENOTFOUND	The name to get was found in the import list, but IPC returned an error which indicates the message was rejected because station may be down. The object name is removed from the import list.
EBADTYPE	Illegal object type argument.
ENOSPACE	No memory available; no value returned.
EBADNAME	Invalid object name.
EBADLEN	Shared variable is too big for data_len.
ENOVALUE	Specified object type has no "value". The caller requested to get the value of an object type process or device.
EBADVREC	"Value" record is corrupt, invalid data type.
EIMPFULL	Import table full; value is returned.
ENOTACTIVE	Not registered with IPC for connectionless service.
EIPCRET	You have an unspecified IPC error.

Call Notes:

1. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects *om_getval* ignores that option. However, the value is still returned.
2. If you get an error return, the Object Manager may still return a value, but it is garbage. Do *not* send the returned value to useful data areas unless there is no error return.
3. If you do not make data_len large enough, value will be too small to hold the returned value. In such a case `om_getval` returns as much data in value as will fit.
4. If data_len is too large, value will contain unused bytes. *om_getval* does not close-up this unused memory space.
5. Status, data_len, and value are modified regardless of error return.

2.4 om_set_confirm – Set the Value of an Object

om_set_confirm sets the value of the specified object and waits for confirmation that it was set. This call is a functional superset of the *set_confirm* call. It uses a PSAP argument to specify the station to which the SETVAL message is sent. Use of the argument eliminates the need to broadcast the SETVAL message. This call works for shared variables, process variables, and aliases. *om_set_confirm* is synchronous; it initiates the SETVAL message and waits for a response before returning control to your process/task.

Format

```
int om_set_confirm(<name>,<obj_type>,<import>,<value>,<status>,<data_len>,
<psap_ptr>)

    char            *name, *value;
    int             obj_type, import, data_len;
    unsigned int    *status;
    PSAP_ADDR       *psap_ptr;
```

Where:

`*name`	– A pointer to the name of the object.
`obj_type`	The named object's type; either VARIABLE or ALIAS.
`import`	1 = Place the object's name on the import list. 0 = Do not place the object's name on the import list.
`*value`	A pointer to the value's location. Specify its size in bytes using `data_len` if it is a string or process variable.
`*status`	A pointer to the location where you have stored the object's data type (range 1-15). An ALIAS object type can only have STRING data type. The shared VARIABLE object type can have a data type of CHARACTER, INTEGER, FLOAT, STRING, OM_BOOL (byte with True/False values only), OM_LNG_INT (long integer), OM_S_PKBOL (short packed boolean), or OM_L_PKBOL (long packed boolean). The Object Manager does not restrict process VARIABLE data types. All shared VARIABLE data types are supported plus CIO_SHORT (unsigned character).
`data_len`	The length (in bytes) allocated for "value". This is needed for STRING data types and all process variables (0 defaults to size of float).
`*psap_ptr`	A pointer to the object's station PSAP address. A value of (PSAP_ADDR *) NULL indicates that no PSAP has been provided (the call acts exactly the same as a *set_confirm* call).

Return Codes:

`OM_SUCCESS`	The set request has been initiated successfully.
`EBADTYPE`	The specified data type is not supported.

	The specified data type does not match the actual data type (shared variables only).
	For ALIAS, the data type is not STRING.
	For shared VARIABLE data type OM_BOOL, specified value is not 1 (True) or 0 (False).
EBADNAME	Object name too long.
	Process variable name with object type not equal to VARIABLE.
ESTRLEN	For STRING data type, specified string length too long or < = 0.
	For STRING data type, specified string length longer than actual string length.
ENOVALUE	Object type DEVICE or PROCESS was specified.
ESECURE	Variable is secured.
EBADVREC	Shared variable value record bad.
ENOCONFIRM	Object not found.
ENOTFOUND	Object not found in specified station. A station is specified by import table entry, PSAP parameter, or list ID and index parameters.
ENOTACTIVE	Caller not activated with IPC.
ENOSPACE	System resource availability problem.
EIPCRET	IPC returned an unspecified error.

Return Code Notes:

1. OM error codes are negative (refer to OM_ECODE.H or Appendix B).
2. Control and I/O (compound processor) returns positive error codes (refer to the document *System Messages*).

Call Notes:

1. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects *om_set_confirm* ignores that option.
2. If a value is successfully set, its BAD and OOS status bits are automatically reset.

2.5 om_setval – Set the Value of an Object

om_setval sets the value of the specified object. This call is a functional superset of the *setval* call. It provides a PSAP argument to specify the station to which the *om_setval* message is to be sent. Use of the PSAP argument eliminates the need to broadcast the SETVAL message. This call works for shared variables, process variables, and aliases. *om_setval* is asynchronous if the object is not in the local station; it initiates the SETVAL message and returns control to your process/task.

Format:

```
int om_setval (<name>,<obj_type>,<import>,<value>,<status>,<data_len>,
<psap_ptr>)

    char            *name, *value;
    int             obj_type, import, data_len;
    unsigned int    *status;
    PSAP_ADDR       *psap_ptr;
```

Where:

`*name`	A pointer to the name of the object.
`obj_type`	The named object's type; either VARIABLE or ALIAS.
`import`	1 = Place the object's name on the import list. 0 = Do not place the object's name on the import list.
`*value`	A pointer to the value's location. Specify its size in bytes using `data_len` if it is a string or process variable.
`*status`	A pointer to the location where you have stored the object's data type (range 1-15). An ALIAS object type can only have STRING data type. The shared VARIABLE object type can have a data type of CHARACTER, INTEGER, FLOAT, STRING, OM_BOOL (byte with True/False values only), OM_LNG_INT (long integer), OM_S_PKBOL (short packed boolean), or OM_L_PKBOL (long packed boolean). The Object Manager does not restrict process VARIABLE data types. All shared VARIABLE data types are supported plus CIO_SHORT (unsigned character).
`data_len`	The length (in bytes) allocated for "value." This is needed for STRING data types and all process variables (0 defaults to size of float).
`*psap_ptr`	A pointer to the object's station PSAP address. A value of (PSAP_ADDR *) NULL indicates that no PSAP has been provided (the call acts exactly the same as a *setval* call).

Return Codes:

`OM_SUCCESS`	The set request has been initiated successfully.
`EBADTYPE`	The specified data type is not supported.

	The specified data type does not match the actual data type (shared variables only). See Note 3.
	For ALIAS, the data type is not STRING.
	For shared VARIABLE data type OM_BOOL, specified value is not 1 (True) or 0 (False). See Note 3.
EBADNAME	Object name too long.
	Process variable name with object type not equal to VARIABLE.
ESTRLEN	For STRING data type, specified string length too long or < = 0.
	For STRING data type, specified string length longer than actual string length. See Note 3.
ENOVALUE	Object type DEVICE or PROCESS was specified.
ESECURE	VARIABLE is secured. See Note 3.
EBADVREC	Shared variable value record bad. See Note 3.

Return Code Notes:

1. OM error codes are negative (refer to OM_ECODE.H or Appendix B).
2. Control and I/O (compound processor) returns positive error codes (refer to the document *System Messages*).
3. These error cases are returned only if the variable is local.

Call Notes:

1. Since this call is asynchronous, it cannot wait for return codes from remote stations. Thus all returns concerning the object to be set can only come from the local station. Control variables can only be local in a Personal Workstation (a PW-C or PW-FB).
2. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects *om_setval* ignores that option.
3. If a value is successfully set, its BAD and OOS status bits are automatically reset.

2.6 set_confirm – Set the Value of an Object

set_confirm sets the value of the specified object and waits for confirmation that it was set. This call works for shared variables, process variables, and aliases. *set_confirm* is synchronous; it initiates the SETVAL message and waits for a response before returning control to your process/task.

Format:

```
int set_confirm(<name>,<obj_type>,<import>,<value>,<status>,<data_len>)
    char            *name, *value;
    int             obj_type, import, data_len;
    unsigned int    *status;
```

Where:

`*name`	A pointer to the name of the object.
`obj_type`	The named object's type; either VARIABLE or ALIAS.
`import`	1 = Place the object's name on the import list. 0 = Do not place the object's name on the import list.
`*value`	A pointer to the value's location. Specify its size in bytes using `data_len` if it is a string or process variable.
`*status`	A pointer to the location where you have stored the object's data type (range 1-15). An ALIAS object type can only have STRING data type. The shared VARIABLE object type can have a data type of CHARACTER, INTEGER, FLOAT, STRING, OM_BOOL (byte with True/False values only), OM_LNG_INT (long integer), OM_S_PKBOL (short packed boolean), or OM_L_PKBOL (long packed boolean). The Object Manager does not restrict process VARIABLE data types. All shared VARIABLE data types are supported plus CIO_SHORT (unsigned character).
`data_len`	The length (in bytes) allocated for "value". This is needed for STRING data types and all process variables (0 defaults to size of float).

Return Codes:

`OM_SUCCESS`	The set request has been initiated successfully.
`EBADTYPE`	The specified data type is not supported. The specified data type does not match the actual data type (shared variables only). For ALIAS, the data type is not STRING. For shared VARIABLE data type OM_BOOL, specified value is not 1 (True) or 0 (False).
`EBADNAME`	Object name too long. Process variable name with object type not equal to VARIABLE.

ESTRLEN	For STRING data type, specified string length too long or < = 0.
	For STRING data type, specified string length longer than actual string length.
ENOVALUE	Object type DEVICE or PROCESS was specified.
ESECURE	Variable is secured. See Note 3.
EBADVREC	Shared variable value record bad.
ENOCONFIRM	Object not found.
ENOTFOUND	Object not found in specified station. A station is specified by import table entry, PSAP parameter, or list ID and index parameters.
ENOTACTIVE	Caller not activated with IPC.
ENOSPACE	System resource availability problem.
EIPCRET	IPC returned an unspecified error.

Return Code Notes:

1. OM error codes are negative (refer to OM_ECODE.H or Appendix B).
2. Control and I/O (compound processor) returns positive error codes (refer to the document *System Messages*).

Call Notes:

1. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects *set_confirm* ignores that option.
2. If a value is successfully set, its BAD and OOS status bits are automatically reset.

2.7 set_cnf_list – Set the Value of an Object

set_cnf_list sets the value of the specified object. This call is a variation of the *om_set_confirm* call. It differs only in the way the PSAP is specified.

The ID of a currently open (optimized or unoptimized) list and the index of a variable entry in the list are specified. The variable must be connected. It is assumed that the list variable and the target object reside in the same station. The PSAP associated with that variable identifies the station to which the SETVAL message is sent. *set_cnf_list* is synchronous; it suspends your task until the transfer is complete.

Format:

```
int set_cnf_list(<name>,<obj_type>,<import>,<value>,<status>,<data_len>,
<open_id>,<ov_index>)

    char          *name, *value;
    int           obj_type, import, data_len, open_id, ov_index;
    unsigned int  *status;
```

Where:

`*name`	A pointer to the name of the object.
`obj_type`	The named object's type; either VARIABLE or ALIAS.
`import`	1 = Place the object's name on the import list. 0 = Do not place the object's name on the import list.
`*value`	A pointer to the value's location. Specify its size in bytes using `data_len` if it is a string or process variable.
`*status`	A pointer to the location where you have stored the object's data type (range 1-15). An ALIAS object type can only have STRING data type. The shared VARIABLE object type can have a data type of CHARACTER, INTEGER, FLOAT, STRING, OM_BOOL (byte with True/False values only), OM_LNG_INT (long integer), OM_S_PKBOL (short packed boolean), or OM_L_PKBOL (long packed boolean). The Object Manager does not restrict process VARIABLE data types. All shared VARIABLE data types are supported plus CIO_SHORT (unsigned character).
`data_len`	The length (in bytes) allocated for "value". This is needed for STRING data types and all process variables (0 defaults to size of float).
`open_id`	ID of open variables list that contains the variable to set.
`ov_index`	Index of a variable entry within the list.

Return Codes:

`OM_SUCCESS`	The set request has been initiated successfully.
`EBADTYPE`	The specified data type is not supported.

	The specified data type does not match the actual data type (shared variables only).
	For ALIAS, the data type is not STRING.
	For shared VARIABLE data type OM_BOOL, specified value is not 1 (True) or 0 (False).
EBADNAME	Object name too long.
	Process variable name with object type not equal to VARIABLE.
ESTRLEN	For STRING data type, specified string length too long or < = 0.
	For STRING data type, specified string length longer than actual string length.
ENOVALUE	Object type DEVICE or PROCESS was specified.
ESECURE	Variable is secured. See Note 3.
EBADVREC	Shared variable value record bad.
ENOCONFIRM	Object not found.
ENOTFOUND	Object not found in specified station. A station is specified by import table entry, PSAP parameter, or list ID and index parameters.
ENOTACTIVE	Caller not activated with IPC.
ENOSPACE	System resource availability problem.
EIPCRET	IPC returned an unspecified error.
ENOTOPENED	Specified list not open.
EBADINDEX	Specified index greater than size of list. Specified index entry not connected.

Return Code Notes:

1. OM error codes are negative (refer to OM_ECODE.H or Appendix B).
2. Control and I/O (compound processor) returns positive error codes (refer to the document *System Messages*).

Call Notes:

1. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects *set_cnf_list* ignores that option.
2. If a value is successfully set, its BAD and OOS status bits are automatically reset.

2.8 setval – Set the Value of an Object

setval sets the value of the specified object. This call works for shared variables, process variables, and aliases. *setval* is asynchronous if the object is not in the local station; it initiates the SETVAL message and returns control to your process/task.

Format:

```
int setval(<name>,<obj_type>,<import>,<value>,<status>,<data_len>)
    char          *name, *value;
    int           obj_type, import, data_len;
    unsigned int  *status;
```

Where:

`*name`	A pointer to the name of the object.
`obj_type`	The named object's type; either VARIABLE or ALIAS.
`import`	1 = Place the object's name on the import list. 0 = Do not place the object's name on the import list.
`*value`	A pointer to the value's location. Specify its size in bytes using `data_len` if it is a string or process variable.
`*status`	A pointer to the location where you have stored the object's data type (range 1-15). An ALIAS object type can only have STRING data type. The shared VARIABLE object type can have a data type of CHARACTER, INTEGER, FLOAT, STRING, OM_BOOL (byte with True/False values only), OM_LNG_INT (long integer), OM_S_PKBOL (short packed boolean), or OM_L_PKBOL (long packed boolean). The Object Manager does not restrict process VARIABLE data types. All shared VARIABLE data types are supported plus CIO_SHORT (unsigned character).
`data_len`	The length (in bytes) allocated for "value". This is needed for STRING data types and all process variables (0 defaults to size of float).

Return Codes:

`OM_SUCCESS`	The set request has been initiated successfully.
`EBADTYPE`	The specified data type is not supported. The specified data type does not match the actual data type (shared variables only). See Note 3. For ALIAS, the data type is not STRING. For shared VARIABLE data type OM_BOOL, specified value is not 1 (True) or 0 (False). See Note 3.
`EBADNAME`	Object name too long. Process variable name with object type not equal to VARIABLE.

ESTRLEN	For STRING data type, specified string length too long or < = 0.
	For STRING data type, specified string length longer than actual string length. See Note 3.
ENOVALUE	Object type DEVICE or PROCESS was specified.
ESECURE	VARIABLE is secured. See Note 3.
EBADVREC	Shared variable value record bad. See Note 3.

Return Code Notes:

1. OM error codes are negative (refer to OM_ECODE.H or Appendix B).
2. Control and I/O (Compound Processor) returns positive error codes (refer to the document *System Messages*).
3. These error cases are returned only if the variable is local.

Call Notes:

1. Since this call is asynchronous, it cannot wait for return codes from remote stations. Thus all returns concerning the object to be set can only come from the local station. Control variables can only be local in a Personal Workstation (a PW-C or PW-FB).
2. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects *setval* ignores that option.
3. If a value is successfully set, its BAD and OOS status bits are automatically reset.

2.9 setval_list – Set Value of an Object

setval_list sets the value of the specified object. This call is a variation of the *om_setval* call. It differs only in the way the PSAP is specified.

The ID of a currently open (optimized or unoptimized) list and the index of a variable entry in the list are specified. The variable must be connected. It is assumed that the list variable and the target object reside in the same station. The PSAP associated with that variable identifies the station to which the SETVAL message is sent. *setval_list* is asynchronous; it initiates the SETVAL message and returns control to your process/task.

Format

```
int setval_list(<name>,<obj_type>,<import>,<value>,<status>,<data_len>,
<open_id>,<ov_index>)
    char          *name, *value;
    int           obj_type, import, data_len, open_id, ov_index;
    unsigned int  *status;
```

Where:

`*name`	A pointer to the name of the object.
`obj_type`	The named object's type; either VARIABLE or ALIAS.
`import`	1 = Place the object's name on the import list. 0 = Do not place the object's name on the import list.
`*value`	A pointer to the value's location. Specify its size in bytes using `data_len` if it is a string or process variable.
`*status`	A pointer to the location where you have stored the object's data type (range 1-15). An ALIAS object type can only have STRING data type. The shared VARIABLE object type can have a data type of CHARACTER, INTEGER, FLOAT, STRING, OM_BOOL (byte with True/False values only), OM_LNG_INT (long integer), OM_S_PKBOL (short packed boolean), or OM_L_PKBOL (long packed boolean). The Object Manager does not restrict process VARIABLE data types. All shared VARIABLE data types are supported plus CIO_SHORT (unsigned character).
`data_len`	The length (in bytes) allocated for "value." This is needed for STRING data types and all process variables (0 defaults to size of float).
`open_id`	ID of open variables list that contains the variable to set.
`ov_index`	Index of a variable entry within the list.

Return Codes:

`OM_SUCCESS`	The set request has been initiated successfully.
`EBADTYPE`	The specified data type is not supported.

	The specified data type does not match the actual data type (shared variables only). See Note 3.
	For ALIAS, the dam type is not STRING.
	For shared VARIABLE data type OM_BOOL, specified value is not 1 (True) or 0 (False). See Note 3.
EBADNAME	Object name too long.
	Process variable name with object type not equal to VARIABLE.
ESTRLEN	For STRING data type, specified string length too long or < = 0.
	For STRING data type, specified string length longer than actual string length. See Note 3.
ENOVALUE	Object type DEVICE or PROCESS was specified.
ESECURE	VARIABLE is secured. See Note 3.
EBADVREC	Shared variable value record bad. See Note 3.
ENOTOPENED	Specified list not open.
EBADINDEX	Specified index greater than size of list. Specified index entry not connected.

Return Code Notes:

1. OM error codes are negative (refer to OM_ECODE.H or Appendix B).
2. Control and I/O (compound_processor) returns positive error codes (refer to the document *System Messages*).
3. These error cases are returned only if the variable is local.

Call Notes:

1. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects *setval_list* ignores that option.
2. If a value is successfully set, its BAD and OOS status bits are automatically reset.

2.10 st_omset_confirm – Set the Value and/or Status of an Object

st_omset_confirm sets the value and/or the status of the specified object. This call is a functional extension of the om_*set_confirm* call. A status mask and a status data parameter allow the specification of object status bits to be set or reset. A status only parameter allows the writing of status without an accompanying data value. This call works for shared variables, process variables, and aliases. *st_omset_confirm* is synchronous; it initiates the SETVAL message and waits for a response before returning control to your process/task.

Format

```
int st_omset_confirm(<name>,<obj_type>,<import>,<value>,<status>,
<data_len>,<psap_ptr>,<st_data>,<st_mask>,<st_only>,<appl_work>)
    char            *name, *value, *appl_work;
    int             obj_type, import, data_len;
    unsigned int    *status, st_data, st_mask, st_only;
    PSAP_ADDR       *psap_ptr;
```

Where:

`*name`	A pointer to the name of the object.
`obj_type`	The named object's type; either VARIABLE or ALIAS.
`import`	1 = Place the object's name on the import list. 0 = Do not place the object's name on the import list.
`*value`	A pointer to the value's location. Specify its size in bytes using `data_len` if it is a string or process variable.
`*status`	A pointer to the location where you have stored the object's data type (range 1-15). An ALIAS object type can only have STRING data type. The shared VARIABLE object type can have a data type of CHARACTER, INTEGER, FLOAT, STRING, OM_BOOL (byte with True/False values only), OM_LNG_INT (long integer), OM_S_PKBOL (short packed boolean), or OM_L_PKBOL (long packed boolean). The Object Manager does not restrict process VARIABLE data types. All shared VARIABLE data types are supported plus CIO_SHORT (unsigned character).
`data_len`	The length (in bytes) allocated for "value". This is needed for STRING data types and all process variables (0 defaults to size of float).
`*psap_ptr`	A pointer to the object's station PSAP address. A value of (PSAP_ADDR *) NULL indicates that no PSAP has been provided (the call acts exactly the same as a *set_confirm* call).
`st_data`	Object status data bits. This word specifies the desired setting for the bits specified in st_mask. Only bits defined in om_udef.h (BAD, OOS, and for process control variables only ACK) are allowed to be set/reset.

st_mask	Object status mask bits. Bits set in this word specify the bits to set/reset according to their setting in the data word. If st_mask is NULL, the call performs like a *serval* call. If st_mask is NULL, the call automatically resets BAD and OOS if a successful value set is performed.
st_only	If st_only is TRUE (non-zero), only a status set is performed. The data value is not set. If st_only is TRUE and st_mask is NULL, the call is a NOP.
appl_work	A pointer to a workstation ID string. For application programs, specify OM_APPLICATION (defined in om_udef.h).

Return Codes:

OM_SUCCESS	The set request has been initiated successfully.
EBADTYPE	The specified data type is not supported.
	The specified data type does not match the actual data type (shared variables only).
	For ALIAS, the data type is not STRING.
	For shared VARIABLE data type OM_BOOL, specified value is not 1 (True) or 0 (False).
EBADNAME	Object name too long.
	Process variable name with object type not equal to VARIABLE.
ESTRLEN	For STRING data type, specified string length too long or < = 0.
	For STRING data type, specified string length longer than actual string length.
ENOVALUE	Object type DEVICE or PROCESS was specified.
ESECURE	Variable is secured. See Note 3.
EBADVREC	Shared variable value record bad.
ENOCONFIRM	Object not found.
ENOTFOUND	Object not found in specified station. A station is specified by import table entry, PSAP parameter, or list ID and index parameters.
ENOTACTIVE	Caller not activated with IPC.
ENOSPACE	System resource availability problem.
EIPCRET	IPC returned an unspecified error.
EBADMSK	Invalid bits set in write status mask. Valid bits are defined in om_udef.h.
EBADWKSTA	Workstation ID name invalid length (must be exactly 13 characters).

Return Code Notes:

1. OM error codes are negative (refer to OM_ECODE.H or Appendix B).
2. Control and I/O (compound processor) returns positive error codes (refer to the document *System Messages*).

3. These error cases returned only if the variable is local.

Call Notes:

1. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects *st_omset_confirm* ignores that option.
2. For process variables, BAD and OOS are always cleared before the masked status write is performed.

2.11 st_om_setval – Set the Value and/or Status of an Object

st_om_setval sets the value and/or the status of the specified object. This call is a functional extension of the *om_setval* call. A status mask and a status data parameter allow the specification of object status bits to be set or reset. A status only parameter allows the writing of status without an accompanying data value. This call works for shared variables, process variables, and aliases. *st_om_setval* is asynchronous if the object is not in the local station; it initiates the SETVAL message and returns control to your process/task.

Format

```
int st_om_setval(<name>,<obj_type>,<import>,<value>,<staus>,<data_len>,
<psap_ptr>,<st_data>,<st_mask>,<st_only>,<appl_work>) -
    char            *name, *value, *appl_work;
    int             obj_type, import, data_len;
    unsigned int    *status, st_data, st_mask, st_only;
    PSAP_ADDR       *psap_ptr;
```

Where:

`*name`	A pointer to the name of the object.
`obj_type`	The named object's type; either VARIABLE or ALIAS.
`import`	1 = Place the object's name on the import list. 0 = Do not place the object's name on the import list.
`*value`	A pointer to the value's location. Specify its size in bytes using `data_len` if it is a string or process variable.
`*status`	A pointer to the location where you have stored the object's data type (range 1-15). An ALIAS object type can only have STRING data type. The shared VARIABLE object type can have a data type of CHARACTER, INTEGER, FLOAT, STRING, OM_BOOL (byte with True/False values only), OM_LNG_INT (long integer), OM_S_PKBOL (short packed boolean), or OM_L_PKBOL (long packed boolean). The Object Manager does not restrict process VARIABLE data types. All shared VARIABLE data types are supported plus CIO_SHORT (unsigned character).
`data_len`	The length (in bytes) allocated for "value." This is needed for STRING data types and all process variables (0 defaults to size of float).
`*psap_ptr`	A pointer to the object's station PSAP address. A value of (PSAP_ADDR *) NULL indicates that no PSAP has been provided (the call acts exactly the same as a *setval* call).
`st_data`	Object status data bits. This word specifies the desired setting for the bits specified in st_mask. Only bits defined in om_udef.h (BAD, OOS, and for process control variables only ACK) are allowed to be set/reset.

st_mask	Object status mask bits. Bits set in this word specify the bits to set/reset according to their setting in the data word. If st_mask is NULL, the call performs like a *serval* call. If st_mask is NULL, the call automatically resets BAD and OOS if a successful value set is performed.
st_only	If st_only is TRUE (non-zero), only a status set is performed. The data value is not set. If st_only is TRUE and st_mask is NULL, the call is a NOP.
appl_work	A pointer to a workstation ID string. For application programs, specify OM_APPLICATION (defined in om_udef.h).

Return Codes:

OM_SUCCESS	The set request has been initiated successfully.
EBADTYPE	The specified data type is not supported.
	The specified data type does not match the actual data type (shared variables only). See Note 3.
	For ALIAS, the data type is not STRING.
	For shared VARIABLE data type OM_BOOL, specified value is not 1 (True) or 0 (False). See Note 3.
EBADNAME	Object name too long.
	Process variable name with object type not equal to VARIABLE.
ESTRLEN	For STRING data type, specified string length too long or < = 0.
	For STRING data type, specified string length longer than actual string length. See Note 3.
ENOVALUE	Object type DEVICE or PROCESS was specified.
ESECURE	VARIABLE is secured.
EBADVREC	Shared variable value record bad. See Note 3.
EBADMSK	Invalid bits set in write status mask. Valid bits are defined in om_udef.h.
EBADWKSTA	Workstation ID name invalid length (must be exactly 13 characters).

Return Code Notes:

1. OM error codes are negative (refer to OM_ECODE.H or Appendix B).
2. Control and I/O (compound processor) returns positive error codes (refer to the document *System Messages*).
3. These error cases are returned only if the variable is local.
4. If a value is successfully set, its BAD and OOS status bits are automatically reset.

Call Notes:

1. Since this call is asynchronous, it cannot wait for return codes from remote stations. Thus all returns concerning the object to be set can only come from the local station. Control variables can only be local in a Personal Workstation (a PW-C or PW-FB).

2. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects *st_om_setval* ignores that option.
3. For process control variables, BAD and OOS are always cleared before the masked status write is performed.

2.12 st_setcnf – Set the Value and/or Status of an Object

st_setcnf sets the value and/or the status of the specified object. This call is a functional extension of the *set_confirm* call. A status mask and a status data parameter allow the specification of object status bits to be set or reset. A status only parameter allows the writing of status without an accompanying data value. This call works for shared variables, process variables, and aliases. *st_setcnf* is synchronous; it initiates the SETVAL message and waits for a response before returning control to your process/task.

Format:

```
int st_setcnf(<name>,<obj_type>,<import>,<value>,<status>,<data_len>,
<st_data>,<st_mask>,<st_only>,<appl_work>)

    char            *name, *value, *appl_work;
    int             obj_type, import, data_len;
    unsigned int    *status, st_data, st_mask, st_only;
```

Where:

`*name`	A pointer to the name of the object.
`obj_type`	The named object's type; either VARIABLE or ALIAS.
`import`	1 = Place the object's name on the import list. 0 = Do not place the object's name on the import list.
`*value`	A pointer to the value's location. Specify its size in bytes using `data_len` if it is a string or process variable.
`*status`	A pointer to the location where you have stored the object's data type (range 1-15). An ALIAS object type can only have STRING data type. The shared VARIABLE object type can have a data type of CHARACTER, INTEGER, FLOAT, STRING, OM_BOOL (byte with True/False values only), OM_LNG_INT (long integer), OM_S_PKBOL (short packed boolean), or OM_L_PKBOL (long packed boolean). The Object Manager does not restrict process VARIABLE data types. All shared VARIABLE data types are supported plus CIO_SHORT (unsigned character).
`data_len`	The length (in bytes) allocated for "value". This is needed for STRING data types and all process variables (0 defaults to size of float).
`st_data`	Object status data bits. This word specifies the desired setting for the bits specified in st_mask. Only bits defined in om_udef.h (BAD, OOS, and for process control variables only ACK) are allowed to be set/reset.
`st_mask`	Object status mask bits. Bits set in this word specify the bits to set/reset according to their setting in the data word. If st_mask is NULL, the call performs like a *setval* call. If st_mask is NULL, the call automatically resets BAD and OOS if a successful value set is performed.

st_only	If st_only is TRUE (non-zero), only a status set is performed. The data value is not set. If st_only is TRUE and st_mask is NULL, the call is a NOP.
appl_work	A pointer to a workstation ID string. For application programs, specify OM_APPLICATION (defined in om_udef.h).

Return Codes:

OM_SUCCESS	The set request has been initiated successfully.
EBADTYPE	The specified data type is not supported.
	The specified data type does not match the actual data type (shared variables only).
	For ALIAS, the data type is not STRING.
	For shared VARIABLE data type OM_BOOL, specified value is not 1 (True) or 0 (False).
EBADNAME	Object name too long.
	Process variable name with object type not equal to VARIABLE.
ESTRLEN	For STRING data type, specified string length too long or < = 0.
	For STRING data type, specified string length longer than actual string length.
ENOVALUE	Object type DEVICE or PROCESS was specified.
ESECURE	Variable is secured. See Note 3.
EBADVREC	Shared variable value record bad.
ENOCONFIRM	Object not found.
ENOTFOUND	Object not found in specified station. A station is specified by import table entry, PSAP parameter, or list ID and index parameters. ENOTFOUND occurs when the object is on the import list, but the station listed as its home cannot find it. This means that the object has probably been moved to another station.
ENOTACTIVE	Caller not activated with IPC.
ENOSPACE	System resource availability problem.
EIPCRET	IPC returned an unspecified error.
EBADMSK	Invalid bits set in write status mask. Valid bits are defined in OM_UDEF.H.
EBADWKSTA	Workstation ID name invalid length (must be exactly 13 characters).

Return Code Notes:

1. OM error codes are negative (refer to OM_ECODE.H or Appendix B).
2. Control and I/O (compound processor) returns positive error codes (refer to the document *System Messages*).
3. These error cases are returned only if the variable is local.

Call Notes:

1. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects *st_setcnf* ignores that option.
2. For process control variables, BAD and OOS are always cleared before the masked status write is performed.

2.13 st_setlist_confirm – Set the Value and/or Status of an Object

st_setlist_confirm sets the value and/or the status of the specified object. This call is a functional extension of the *set_cnf_list* call. A status mask and a status data parameter allow the specification of object status bits to be set or reset. A status only parameter allows the writing of status without an accompanying data value. This call works for shared variables, process variables, and aliases. *st_setlist_confirm* is synchronous; it initiates the SETVAL message and waits for a response before returning control to your process/task.

Format:

```
int st_setlist_confirm(<name>,<obj_type>,<import>,<value>,<status>,
<data_len>,<open_id>,<ov_index>,<st_data>,<st_mask>,<st_only>,<appl_work>)
    char          *name, *value, *appl_work;
    int           obj_type, import, data_len, open_id, ov_index;
    unsigned int  *status, st_data, st_mask, st_only;
```

Where:

`*name`	A pointer to the name of the object.
`obj_type`	The named object's type; either VARIABLE or ALIAS.
`import`	1 = Place the object's name on the import list. 0 = Do not place the object's name on the import list.
`*value`	A pointer to the value's location. Specify its size in bytes using `data_len` if it is a string or process variable.
`*status`	A pointer to the location where you have stored the object's data type (range 1-15). An ALIAS object type can only have STRING data type. The shared VARIABLE object type can have a data type of CHARACTER, INTEGER, FLOAT, STRING, OM_BOOL (byte with True/False values only), OM_LNG_INT (long integer), OM_S_PKBOL (short packed boolean), or OM_L_PKBOL (long packed boolean). The Object Manager does not restrict process VARIABLE data types. All shared VARIABLE data types are supported plus CIO_SHORT (unsigned character).
`data_len`	The length (in bytes) allocated for "value". This is needed for STRING data types and all process variables (0 defaults to size of float).
`open_id`	ID of open variables list that contains the variable to set.
`ov_index`	Index of a variable entry within the list.
`st_data`	Object status data bits. This word specifies the desired setting for the bits specified in st_mask. Only bits defined in om_udef.h (BAD, OOS, and for process control variables only ACK) are allowed to be set/reset.
`st_mask`	Object status mask bits. Bits set in this word specify the bits to set/reset according to their setting in the data word. If st_mask is NULL, the call

	performs like a *serval* call. If st_mask is NULL, the call automatically resets BAD and OOS if a successful value set is performed.
st_only	If st_only is TRUE (non-zero), only a status set is performed. The data value is not set. If st_only is TRUE and st_mask is NULL, the call is a NOP.
appl_work	A pointer to a workstation ID string. For application programs, specify OM_APPLICATION (defined in om_udef.h).

Return Codes:

OM_SUCCESS	The set request has been initiated successfully.
EBADTYPE	The specified data type is not supported.
	The specified data type does not match the actual data type (shared variables only).
	For ALIAS, the data type is not STRING.
	For shared VARIABLE data type OM_BOOL, specified value is not 1 (True) or 0 (False).
EBADNAME	Object name too long.
	Process variable name with object type not equal to VARIABLE.
ESTRLEN	For STRING data type, specified string length too long or < = 0.
	For STRING data type, specified string length longer than actual string length.
ENOVALUE	Object type DEVICE or PROCESS was specified.
ESECURE	Variable is secured.
EBADVREC	Shared variable value record bad.
ENOCONFIRM	Object not found.
ENOTFOUND	Object not found in specified station. A station is specified by import table entry, PSAP parameter, or list ID and index parameters.
ENOTACTIVE	Caller not activated with IPC.
ENOSPACE	System resource availability problem.
EIPCRET	IPC returned an unspecified error.
ENOTOPENED	Specified list not open.
EBADINDEX	Specified index greater than size of list. Specified index entry not connected.
EBADMSK	Invalid bits set in write status mask. Valid bits are defined in om_udef.h.
EBADWKSTA	Workstation ID name invalid length (must be exactly 13 characters).

Return Code Notes:

1. OM error codes are negative (refer to OM_ECODE.H or Appendix B).

2. Control and I/O (compound processor) returns positive error codes (refer to the document *System Messages*).
3. If a value is successfully set, its BAD and OOS status bits are automatically reset.

Call Notes:

1. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects *st_setlist_confirm* ignores that option.
2. For process control variables, BAD and OOS are always cleared before the masked status write is performed.

2.14 st_set_list – Set the Value and/or Status of an Object

st_set_list sets the value and/or the status of the specified object. This call is a functional extension of the *setval_list* call. A status mask and a status data parameter allow the specification of object status bits to be set or reset. A status only parameter allows the writing of status without an accompanying data value. This call works for shared variables, process variables, and aliases. *st_set_list* is asynchronous if the object is not in the local station; it initiates the SETVAL message and returns control to your process/task.

Format:

```
int st_set_list(<name>,<obj_type>,<import>,<value>,<status>,<data_len>,
<open_id>,<ov_index>,<st_data>,<st_mask>,<st_only>,<appl_work>)
    char            *name, *value *appl_work;
    int             obj_type, import, data_len, open_id,ov_index;
    unsigned int    *status, st_data, st_mask, st_only;
```

Where:

`*name`	A pointer to the name of the object.
`obj_type`	The named object's type; either VARIABLE or ALIAS.
`import`	1 = Place the object's name on the import list. 0 = Do not place the object's name on the import list.
`*value`	A pointer to the value's location. Specify its size in bytes using `data_len` if it is a string or process variable.
`*status`	A pointer to the location where you have stored the object's data type (range 1-15). An ALIAS object type can only have STRING data type. The shared VARIABLE object type can have a data type of CHARACTER, INTEGER, FLOAT, STRING, OM_BOOL (byte with True/False values only), OM_LNG_INT (long integer), OM_S_PKBOL (short packed boolean), or OM_L_PKBOL (long packed boolean). The Object Manager does not restrict process VARIABLE data types. All shared VARIABLE data types are supported plus CIO_SHORT (unsigned character).
`data_len`	The length (in bytes) allocated for "value." This is needed for STRING data types and all process variables (0 defaults to size of float).
`open_id`	ID of open variables list that contains the variable to set.
`ov_index`	Index of a variable entry within the list.
`st_data`	Object status data bits. This word specifies the desired setting for the bits specified in st_mask. Only bits defined in om_udef.h (BAD, OOS, and for process control variables only ACK) are allowed to be set/reset.
`st_mask`	Object status mask bits. Bits set in this word specify the bits to set/reset according to their setting in the data word. If st_mask is NULL, the call

performs like a *serval* call. If st_mask is NULL, the call automatically resets BAD and OOS if a successful value set is performed.

st_only	If st_only is TRUE (non-zero), only a status set is performed. The data value is not set. If st_only is TRUE and st_mask is NULL, the call is a NOP.
appl_work	A pointer to a workstation ID string. For application programs, specify OM_APPLICATION (defined in om_udef.h).

Return Codes:

OM_SUCCESS	The set request has been initiated successfully.
EBADTYPE	The specified data type is not supported.
	The specified data type does not match the actual data type (shared variables only). See Note 3.
	For ALIAS, the data type is not STRING.
	For shared VARIABLE data type OM_BOOL, specified value is not 1 (True) or 0 (False). See Note 3.
EBADNAME	Object name too long.
	Process variable name with object type not equal to VARIABLE.
ESTRLEN	For STRING data type, specified string length too long or < = 0.
	For STRING data type, specified string length longer than actual string length. See Note 3.
ENOVALUE	Object type DEVICE or PROCESS was specified.
ESECURE	VARIABLE is secured. See Note 3.
EBADVREC	Shared variable value record bad. See Note 3.
ENOTOPENED	Specified list not open.
EBADINDEX	Specified index greater than size of list. Specified index entry not connected.
EBADMSK	Invalid bits set in write status mask. Valid bits are defined in om_udef.h.
EBADWKSTA	Workstation ID name invalid length (must be exactly 13 characters).

Return Code Notes:

1. OM error codes are negative (refer to OM_ECODE.H or Appendix B).
2. Control and I/O (compound processor) returns positive error codes (refer to the document *System Messages*).
3. These error cases are returned only if the variable is local.

Call Notes:

1. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects *st_set_list* ignores that option.

2. For process control variables, BAD and OOS are always cleared before the masked status write is performed.

2.15 st_setval – Set the Value and/or Status of an Object

st_setval sets the value and/or the status of the specified object. This call is a functional extension of the *setval* call. A status mask and a status data parameter allow the specification of object status bits to be set or reset. A status_only parameter allows the writing of status without an accompanying data value. This call works for shared variables, process variables, and aliases. *st_setval* is asynchronous if the object is not in the local station; it initiates the SETVAL message and returns control to your process/task.

Format

```
int st_setval(<name>,<obj_type>,<import>,<value>,<status>,<data_len>,
<st_data>,<st_mask>,<st_only>,<appl_work>)
    char            *name, *value, *appl_work;
    int             obj_type, import, data_len;
    unsigned int    *status, st_data, st_mask, st_only;
```

Where:

`*name`	A pointer to the name of the object.
`obj_type`	The named object's type; either VARIABLE or ALIAS.
`import`	1 = Place the object's name on the import list. 0 = Do not place the object's name on the import list.
`*value`	A pointer to the value's location. Specify its size in bytes using `data_len` if it is a string or process variable.
`*status`	A pointer to the location where you have stored the object's data type (range 1-15). An ALIAS object type can only have STRING data type. The shared VARIABLE object type can have a data type of CHARACTER, INTEGER, FLOAT, STRING, OM_BOOL (byte with True/False values only), OM_LNG_INT (long integer), OM_S_PKBOL (short packed boolean), or OM_L_PKBOL (long packed boolean). The Object Manager does not restrict process VARIABLE data types. All shared VARIABLE data types are supported plus CIO_SHORT (unsigned character).
`data_len`	The length (in bytes) allocated for "value." This is needed for STRING data types and all process variables (0 defaults to size of float).
st_data	Object status data bits. This word specifies the desired setting for the bits specified in st_mask. Only bits defined in om_udef.h (BAD, OOS, and for process control variables only ACK) are allowed to be set/reset.
st_mask	Object status mask bits. Bits set in this worked specify the bits to set/reset according to their setting in the data word. If st_mask is NULL, the call performs like a *setval* call. If st_mask is NULL, the call automatically resets BAD and OOS if a successful value set is performed.

st_only	If st_only is TRUE (non-zero), only a status set is performed. The data value is not set. If st_only is TRUE and st_mask is NULL, the call is a NOP.
appl_work	A pointer to a workstation ID string. For application programs, specify OM_APPLICATION (defined in om_udef.h).

Return Codes:

OM_SUCCESS	The set request has been initiated successfully.
EBADTYPE	The specified data type is not supported.
	The specified data type does not match the actual data type (shared variables only). See Note 3.
	For ALIAS, the data type is not STRING.
	For shared VARIABLE data type OM_BOOL, specified value is not 1 (True) or 0 (False). See Note 3.
EBADNAME	Object name too long.
	Process variable name with object type not equal to VARIABLE.
ESTRLEN	For STRING data type, specified string length too long or < = 0.
	For STRING data type, specified string length longer than actual string length. See Note 3.
ENOVALUE	Object type DEVICE or PROCESS was specified.
ESECURE	VARIABLE is secured. See Note 3.
EBADVREC	Shared variable value record bad. See Note 3.
EBADMSK	Invalid bits set in write status mask. Valid bits are defined in om_udef.h.
EBADWKSTA	Workstation ID name invalid length (must be exactly 13 characters).

Return Code Notes:

1. OM error codes are negative (refer to OM_ECODE.H or Appendix B).
2. Compound I/O (compound processor) returns positive error codes (refer to the document *System Messages*).
3. These error cases are returned only if the variable is local.

Call Notes:

1. Since this call is asynchronous, it cannot wait for return codes from remote stations. Thus all returns concerning the object to be set can only come from the local station. Control variables can only be local in a Personal Workstation (a PC PW-C or PW-FB).
2. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects *st_setval* ignores that option.
3. For process control variables, BAD and OOS are always cleared before the masked status write is performed.

3. C Calls to Access/Update Sets of Variables

These calls allow your task to establish lists of process variables (using *omopen*) with values indicating how much change has to take place in the variable's value before your task is notified (by *dqchange*).

Copies of the values and status of the variables in an opened list are maintained locally so that the response on reads is immediate and requires no additional network traffic. These calls are designed for tasks needing ongoing access to many process variables.

A task can read from or write to variables on a list returned by *omopen* until you issue an *omclose*. At that time (for optimized lists) the Object Manager returns a table of addresses where it found the variables.

Subsequent *omopens* can use the same table of addresses, thus saving processing time.

The OM C calls to access/update sets of variables are listed in this Section in alphabetical order and summarized in Table 3-1.

Table 3-1. C Calls to Access/Update Sets of Variables

Paragraph No.	Call	Function
3.1	dqchange	Check for object value changes
3.2	dqlist	Dequeue open variables list
3.3	omclose	Close the specified variable list
3.4	omopen	Open a set of variables
3.5	omread	Read values from opened list
3.6	omwrite	Write values to opened list
3.7	omwrstat	Write values and/or status to opened list

3.1 dqchange – Check for Object Value Changes

3.1.1 dqchange call

This call dequeues the notification of change in a variable list which the specified process has opened (with *omopen*). The list is enqueued when a change-driven event occurs to a variable within the list, and the list is not already in the change queue. The updated information is also copied into the list. In order to register as a change, the change has to be greater than or equal to the delta_value (in the *omopen* call) or the status has to change. To be notified, set the NOTIFY option in the list at *omopen* time. If the NOTIFY flag is set, the header node for the open variables list is linked to the end of a change notification queue, and the change notification message is posted to the mailbox for that queue. Change queues are created and managed by the Object Manager on a per process basis. Change notification for a given list is placed on the queue for the process that opened the list. The *dqchange*() caller specifies a process ID, and therefore identifies a particular change queue.

Format:

```
int dqchange(<pid>,<suspend>,<open_id>,<size_list>,<value_list>,_
<ret_size>)

                int             pid, suspend, size_list;
                int             *open_id, *ret_size;
                struct value    *value_list;
```

Where:

`pid`	The process id number of the process that opened the target list.
`suspend`	1 = Suspend execution until a change comes in. 0 = Do not suspend; if no changes, return code = EQEMPTY.
`*open_id`	A pointer to the integer location that will receive the open_id which tells you the first variable list that was found to have changes.
`size_list`	The number of elements in the value structures array that you have created to receive the changed values.
`*value_list`	A pointer to the caller-supplied array of structures. Each new/changed value is returned in one of the array element structures. The array of "value" structures is described below.
`*ret_size`	A pointer to the location where the Object Manager returns the number of changed variables returned.

```
struct value {
        int         index;
        int         status;
        union       var_val {
                        char    letter;
                        int     word;
                        float   fpoint;
```

```
        } uval;
    } values[ ];
```

`index` The index value into the variable list corresponding to the variable that changed.

`status` The data type and status of the variable that changed:

0 = This value not read yet, check for error

Data type – Low byte, bits 0 to 4

1 = CHARACTER	5 = BOOL
2 = INTEGER	6 = OM_LNG_INT or POINTER
3 = FLOAT or REAL	9 = OM_PACKED_BOOL
4 = STRING	10 = OM_LONG_PACKED_BOOL

Error status – Low byte, bits 5 to 7

0 = No response	4 = Bad data type (string) or unconnectable compound
1 = Being scanned	5-6 = (not used)
2 = Disconnected	7 = not sent (error returned)
3 = Deleted	

`uval` The actual value of the variable.

`ret_size` Number of elements returned.

Return Codes:

`OM_SUCCESS`	Changed variables have been successfully returned.
`ESECURE`	The omopen specified write access and attempted to connect to a variable which had been previously secured for write access. None of the variables are connected. The caller should perform an omclose to close the list.
`ENOTOPENED`	The specified process id did not perform any omopens which resulted in leaving a list opened, or the omopen did not specify any variables for notification access.
`ECONNBAD`	The omopen could not establish an IPC connection because no connections were left. You should close the list using the omclose.
`EQNOTEMPTY`	The structure "var" array, supplied by the user, is too small to hold all the changes that the Object Manager has in its queue.
`EQEMPTY`	You did not specify the suspend option and the queue does not contain the specified list.
`OM_ECBSY`	Connection was busy on at least one station. The others are opened. There is no auto-reconnect.

Call Notes:

1. Each variable list is identified by the open_id returned when the list was opened.

2. You can specify any task (process_id) that has open lists. If more than one list has changes, dqchange returns those in the first list, then removes that list from its queue. Use another call to get the changes in the next list.
3. There is a macro, v_varlist(s), that statically or dynamically allocates the value structure for you. This macro is defined in the om_user.h include file:

```
#define v_varlist(s) valloc(sizeof(struct value)*s)
```

The following is an example of how this macro could be used:

```
size_list = 10;
value_list = v_varlist(10);
if (value_list == (struct value *) NULL)
{
        ....
}
```

The first line sets the size_list calling argument. The second line dynamically allocates a 10-element array of "value" structures.
If you use this macro, be sure to free the memory when the structure is no longer needed. It is also advisable to check that the macro did not return a NULL pointer indicating insufficient available memory.

4. A process id number is maintained by the operating system for every task. A task can retrieve its own process id number using the getpid command. For more information on getpid(2) refer to the *UNIX Programmer's Reference Manual*. You can issue a dqchange for the variable list of some other task, but since you can only use getpid for your own task number, the other task has to have sent you its process id.
5. Be careful of specifying that this call suspend. If an omopen failed, and you dqchange with the suspend option, it suspends forever. Check your omopens to make sure they were successful.

3.1.2 Change Queues

The Object Manager has the ability to search a queue, and to dequeue an element regardless of its position within the queue.

The Process/Queue Table is a high level data structure used to manage change queues. This is an array of change queue descriptors dynamically allocated by code, and pointed to by process queue tables.

A change queue descriptor contains the ID of the process or task for which the queue has been allocated. There is no more than one queue per ID, so the ID is passed to *dqchange*() or *dqlist*() to identify a particular change queue.

A process/task with at least one currently open list containing change driven variables has a change queue allocated for it. The queue descriptor contains the current number of change driven lists opened by the process. The queue descriptor is free and may be (re)allocated when the count is zero.

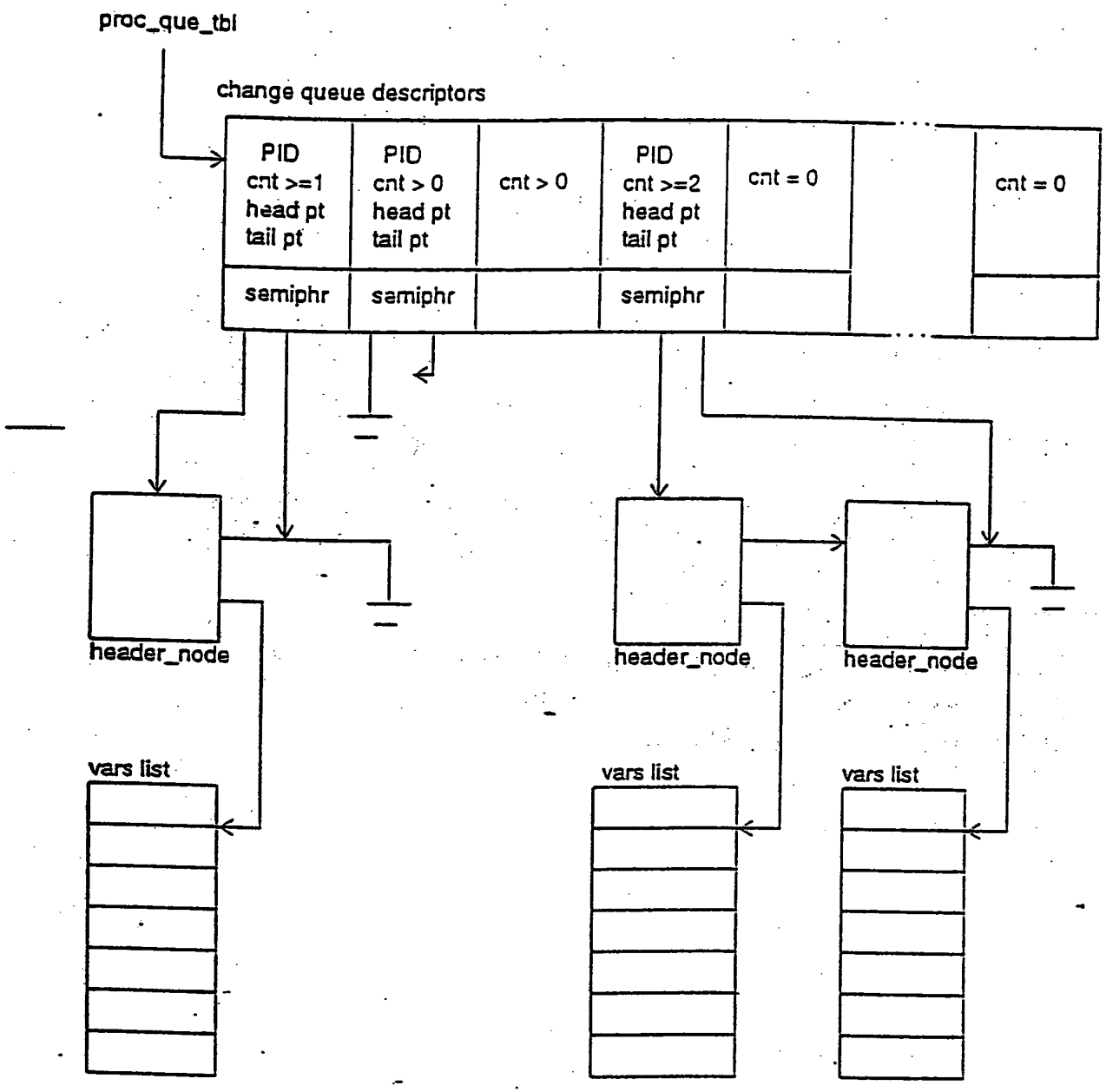
The change queue is a singly linked FIFO queue of open variable list headers. Head and tail pointers are kept in the queue descriptor. The `head_ptr` member points to the first list header in the queue, and the `tail_ptr` member points to the NULL pointer that terminates the linked queue. If the queue is empty, the `head_ptr` contains NULL, and the `tail_ptr` points to the `head_ptr`.

A list header is added to the tail of the queue by storing a pointer to the list header at the location indicated by `tail_ptr`. Then `tail_ptr` is changed to point into the list header, to the NULL pointer terminating the queue.

When *dqchange*() removes a list header from the queue, it is the header to which `head_ptr` points. When *dqlist* removes a header, it is a header that may have been found anywhere within the queue. The header is removed from the queue and pointers are updated. If removing the header leaves the queue empty, the `tail_ptr` must be updated to point to `head_ptr`.

The UNIX semaphore in a change queue descriptor is for use with the system calls. When a queue is empty and the *dqchange*() caller wishes to suspend until the next change notification, it is accomplished using a system call that specifies the semaphore associated in the queue descriptor. When a change occurs, notification is accomplished using a matching call.

The following diagram shows the change queue hierarchy. There are three change queue descriptors in use. This means that there are three processes with open lists of change driven variables. One of the queues is empty. For two of the processes, changes to lists have occurred but have not yet been seen by the application.


proc_que_tbl
change queue descriptors
PID
cnt >=1
head pt
tail pt
PID
cnt > 0
head pt
tail pt
cnt > 0
PID
cnt >=2
head pt
tail pt
cnt = 0
cnt = 0
semiphr
semiphr
semiphr
header_node
header_node
header_node
vars list
vars list
vars list

3.2 dqlist – Dequeue Open Variables List

This call dequeues the notification of change within the specified open variables list. The list is enqueued when a change driven event occurs to a variable within the list, and the list is not already in the change queue. The updated information is copied into the list.

Like *dqchange*, *dqlist* specifies a process ID, and therefore identifies a particular change queue. The list must have been opened by the specified process. No change notifications can be found otherwise. When this call is made, the Object Manager checks the change queue associated with the specified process ID for the change notification for the specified list. If the list is not in the change queue than the call returns EQEMPTY (whether or not the queue is actually empty). Once the change notification for the list is found in the change queue, it is removed from the queue. The value list information is returned to the caller, and the *dqlist*() call returns OM_SUCCESS.

Format:

```
int dqlist(<pid>,<suspend>,<open_id>,<size_list>,<value_list>,<ret_size>)
    int             pid, suspend, size_list;
    int             *open_id, *ret_size;
    struct value    *value_list;
```

Where:

pid	The process id number of the process that opened the target list.
suspend	0 - Must be a zero. Cannot Suspend.
size_list	The number of elements in the values array.
IN *open_id	Integer ID of the target list. The ID is obtained from the previous *omopen* call. The list must currently be open.
*value_list	A pointer to the caller-supplied array of structures. Each new/changed value is returned in one of the array element structures. The array of "value" structures is described below.
*ret_size	A pointer to an integer which receives the number of array elements containing returned variables. .

```
struct value {
        int         index;
        int         status;
        union       var_val {
                            char    letter;
                            int     word;
                            float   fpoint;
        } uval;
} values[ ];
```

index	The index value into the variable list corresponding to the variable that changed.

`status` The data type and status of the variable that changed:

0 = This value not read yet, check for error

Data type – Low byte, bits 0 to 4

1 = CHARACTER	5 = BOOL
2 = INTEGER	6 = OM_LNG_INT or POINTER
3 = FLOAT or REAL	9 = OM_PACKED_BOOL
4 = STRING	10 = OM_LONG_PACKED_BOOL

Error Status –Low byte, bits 5 to 7

0 = No response	4 = Bad data type (string)
1 = Being scanned	or unconnectable compound
2 = Disconnected	5-6 = (not used)
3 = Deleted	7 = not sent (error returned

`uval` The actual value of the variable.

`ret_size` Number of elements returned.

Return Codes:

`OM_SUCCESS` Changed variables have been successfully returned.

`ESECURE` The *omopen* specified write access and attempted to connect to a variable which had been previously secured for write access. None of the variables are connected. The caller should perform an omclose to close the list.

`ENOTOPENED` The specified process id did not perform any omopens which resulted in leaving a list opened, or the omopen did not specify any variables for notification access.

`ECONNBAD` The *omopen* could not establish an IPC connection because no connections were left. You should close the list using the omclose.

`EQNOTEMPTY` The structure "var" array, supplied by the user, is too small to hold all the changes that the Object Manager has in its queue.

`EQEMPTY` You did not specify the suspend option and the queue does not contain the specified list.

Call Notes:

1. Each variable list is identified by the open_id returned when the list was opened.
2. There is a macro, v_varlist(s), that statically or dynamically allocates the value structure for you. This macro is defined in the om_user.h include file:

```
#define v_varlist(s) valloc(sizeof(struct value)*s)
```

The following is an example of how this macro could be used:

```
size_list = 10;
value_list = v_varlist(10);
if (value_list == (struct value *) NULL)
```

```
{
    ....
}
```

The first line sets the size_list calling argument. The second line dynamically allocates a 10-element array of "value" structures.
If you use this macro, be sure to free the memory when the structure is no longer needed. It is also advisable to check that the macro did not return a NULL pointer indicating insufficient available memory.

3. A process id number is maintained by the operating system for every task. A task can retrieve its own process id number using the getpid command. For more information on getpid(2) refer to the *UNIX Programmer Reference Manual*. You can issue a dqlist for the variable list of some other task, but since you can only use getpid for your own task number, the other task has to have sent you its process id.
4. Be careful of specifying that this call suspend. If an omopen failed, and you dqlist with the suspend option = 1, it suspends forever. Check your omopens to make sure they were successful.

3.3 omclose – Close the Specified Variable List

This call closes a list of local or remote variables by releasing the connection established with the omopen call. It also supplies all of the header information, variable list, and address table to speed up future omopen calls to the same list.

omclose is asynchronous; it initiates the request then immediately returns control to your task.

Format:

```
int omclose(<open_id>,<header>,<var_list>,<addr_tbl>)
    int                     open_id;
    struct om_header_node   *header;
    struct open_var         *var_list;
    struct net_addr         *addr_tbl;
```

Where:

`open_id`	The id number returned from the *omopen* call.
`*header`	A pointer to the `om_header_node` structure.
`*var_list`	A pointer to the value structures array.
`*addr_tbl`	A pointer to the address table for the list.

Return Codes:

`OM_SUCCESS`	The request has been initiated.
`ENOTOPEND`	There is no open list corresponding to this open_id.
`ENOTACTIVE`	Not registered with IPC for connectionless service.
`EIPCRET`	You have an unspecified IPC error other than `ENOTACTIVE`.

Call Notes:

1. The header node and variable list (and address table, if optimized) are returned so you can specify them the next time you open this list. When the Object Manager is passed an optimized list with a network address table, it can open the list much more quickly.
2. Do not modify the optimized list structures if the list is to be opened again at a later time.
3. It is imperative that the lists be closed before any exit from the program.

3.4 omopen – Open a Set of Variables

This call opens a list of up to 255 local or remote variables. *omopen* operates on a list of local and/or remote variables, creating a set of connections to the variables. These connections must be made before any data access can be done on the set of variables using the list. Therefore, you must use this call before using any of the other list access calls. There are both optimized and unoptimized versions of *omopen*. This call is asynchronous for local variables; it initiates the request and then immediately returns control to your task. This call is synchronous for remote variables; you must wait until the transmission of remote open messages is complete.

Format:

```
int omopen(<om_descriptor>,<open_id>)
    struct om_header_node      *om_descriptor;
    int                        *open_id;
```

Where:

*om_descriptor	A pointer to the header_node data structure, described in paragraph 3.4.1, Optimized and Unoptimized omopen calls.
*open_id	A pointer to the location where the Object Manager returns the list number.

Return Codes:

OM_SUCCESS	The list has been opened.
EBADRSZ	For an optimized list, being opened the first time, the current size of the Network Address Table is not zero. The list is not opened.
ENOACTIVE	Call not registered with IPC for connectionless communications. List is opened, but remote variables not being scanned.
ENOTFOUND	One or more variables can't be found; the rest are being scanned (unoptimized version only)
ENOQUE	The list specified change notification and there are no change queues available. The list is not opened.
EOMOSIZE	The size of the open variable list exceeds the maximum. The list is not opened.
ESECURE	A connection was attempted to a variable which has previously been connected for write access. If this error is returned, none of the variables in the list are connected. The list is opened but not available for write access.
ENOSPACE	There is not enough room in memory to open a list, try again later.
EOPENED	The list is already opened.
EBADLIST	The variable list pointer is NULL. The list is not opened.
ECONNBAD	List open but not connected, no write access.

ENOADDTBL	List is marked as having been optimized. There is a NULL pointer instead of a Network Address Table. The list is not opened.
ENOSEND	Not enough space to allocate message buffers. List is open, but remotes are not connected or scanned, local variables are being scanned.
ENOADRSP	You specified a Network Address Table that is too small. The list is open but none of the remote variables are being scanned. Variables that fit in the table are being scanned (optimized version only).
EMAXOPNS	The omopen id table is full (maximum number of opens reached); list not opened.
ESCANFUL	List open, but the local scanner database is full; variables that fit in the database are being scanned.
EIPCRET	You have an unspecified IPC error, other than ENOTACTIVE.

Call Notes:

1. Names in the open list should be uppercase to ensure reconnection. All the specified variables are scanned, at the scan_rate, for changes greater than or equal to the delta value. Those that change can be picked up with a dqchange or omread.
2. The unoptimized version of this call saves space by using the OM's address table. Your variable addresses are found and added each time you open the list.
3. Use the open_id to issue the omclose call. It is imperative that the lists be closed before any exit from the program.
4. Optimized is faster, but takes up more memory; unoptimized is slower but uses less memory. The first omopen is always unoptimized (since it has to find the network addresses).
5. For lists with write access only, specify NO_NOTIFY and a large delta. This saves the overhead of unnecessary data reading, and saves a change queue.

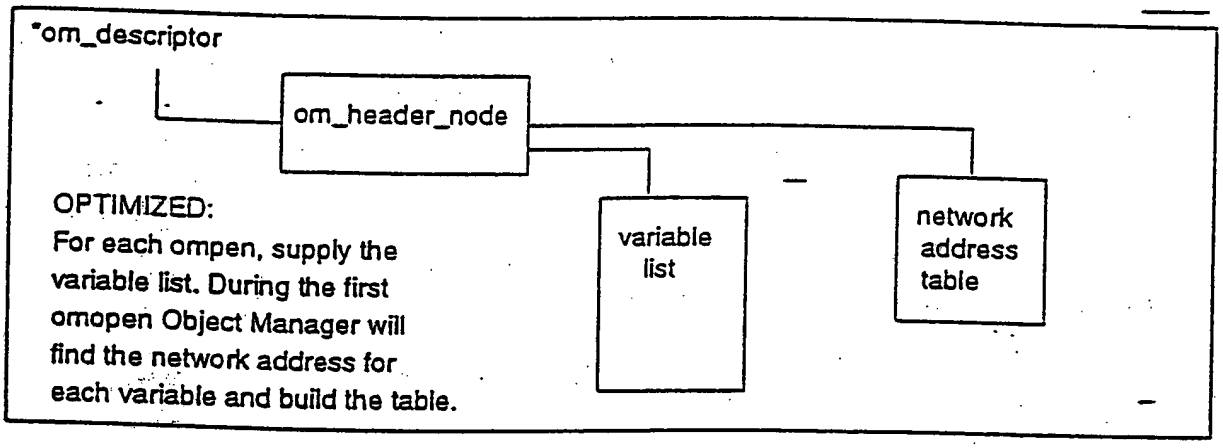
3.4.1 Optimized and Unoptimized omopen calls

Whether an *omopen* call is optimized or unoptimized depends on whether the network address table_pointer (in the `om_header_node` structure) is null or not. For the faster, optimized operation you have to provide a Network Address Table in addition to the `om_header_node` and a list of all the variables you will want to access.

A list which does not have its own Network Address Table is not optimizable. The `om_header_node` contains a NULL pointer instead of a pointer to an address table. When the list is opened, there is no information showing where on the network the variables should be found. Those not found locally will be sought and opened via a broadcast message. Because the list does not have its own address table, the returned network addresses are temporarily recorded in the Object Manager's address table as long as the list is open. (The OM's address table is used by multiple functions within the local station. The advantage is that no duplicate PSAP's need to be stored within the local station.) When the list is closed, the location information is discarded.

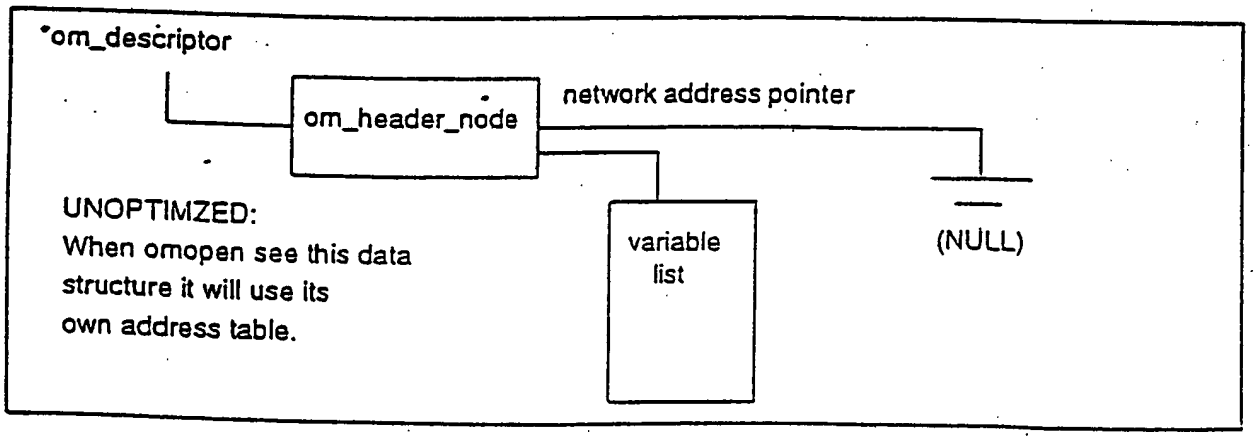
To prepare a list for optimization (in anticipation of repeated *omopen*() calls), you must allocate memory for the address table in the caller's data space. The table may initially contain garbage. The pointer to the table, along with the current (zero) and maximum size values, is placed in the om_header_node. The first time the list is opened, the locations of its variables are not known. Those that are found locally, and those that are found via the broadcast will have their corresponding PSAP's recorded in the Network Address Table belonging to the list. The list is then considered to be optimized. However, it is only partially optimized if any variable was not found by the time the list is closed. The available location information is preserved by the address table (and variable list). The next time the list is opened, this information is used to avoid broadcasting, and makes the processing considerably faster.

The relationships between the structures of open variable lists, both unoptimizable and optimizable, are shown below.



When you call *omclose*, the Object Manager updates the information in the address table, variable list and header node. Save this data, and the next time you *omopen* this list, supply all three to greatly speed up the call. (Do not change any of them!)

For unoptimized operation, which takes up less memory, you just provide header node and a list of variables. The Object Manager will use its own address table. Create the list structure as follows:



3.4.2 omopen User Initialization

Before calling *omopen*(), you must be a registered user and activated with IPC for connectionless communication. The data structures associated with a list must be allocated and initialized before the list can be opened.

The following definitions statically allocate space for an `om_header_node`, a `variable_list` and a Network Address Table. A list with ten elements permits connections to that many variables. A list may contain up to 255 elements or variable address entries. The maximum size (five elements in the example shown below) is based on the number of stations (local and remote) in which any of the listed variables may be found. These structure definitions are defined when the user includes `om_user.h`.

The following C code fragments are examples of how you can set up your data structures:

```
#include <om_user.h>
main ()
{
        static    struct om_header_node    om_descriptor;
        static    struct open_var          variable_list[10];
        static    struct net_adr           net_adr_tbl[5];
        static    int                      open_id;
```

The above example statically allocates a variable list of 10 elements (i.e., you wish to connect to 10 variables). It also allocates a header node and a Network Address Table for five entries.

The C code below initializes the list header node. This is an optimizable list, since the pointer to a Network Address Table, rather than a Null Pointer, is stored in the `net_adr_tbl_ptr` member of the header node. The address table is empty, as indicated by the current size of zero, but can hold up to the specified maximum number of address entries.

The variable list structure is also linked to the header node. The current and maximum sizes of a list are the same unless there are unused entries at the end of the list. Such entries would used with the unoptimized lists. A maximum size of zero is logically equivalent to the specified current size.

List state information is kept in the header node. The list has an access mode of read-only, write-only, or read/write. The access mode codes are defined in `om_user.h` and one code must be placed in the task_status member of the header node.

The `HTSKST_OPM` (optimization change flag) bit of the task status member must be initialized to zero to show that the list is not optimized. (This is accomplished below in the process of storing `OM_R_ACCESS`.)

The scanner update frequency is specified in the list header. The 16-bit scan rate member of the header node is divided into a scan delay high byte and a scan rate low byte. The scan delay byte contains the number of half-seconds between the first and second scanner updates. The scan rate byte contains the number of half-seconds between updates thereafter. A scan delay of zero means that all scanner updates will occur at the specified scan rate.

Undefined bits and fields in the header node are reserved by the Object Manager and must be set to zero. Among these are bits in the task_status and status_ext.

The following code example is for setting up the structures:

```
byte_fill (& om_descriptor, 0, size of(struct om_header_node));
om_descriptor.task_status          = OM_R_ACCESS;
om_descriptor.status_ext           = 0;
om_descriptor.scan_rate            = (0 << 8)! 3;
om_descriptor.open_list_ptr        = variable_list;
om_descriptor.cur_size_open_list   = 10;
om_descriptor.max_size_open_list   = 10;
om_descriptor.net_adr_tbl_ptr      = net_adr_tbl;
om_descriptor.size_net_adr_tbl     = 5;
```

This sets net_adr_tbl_ptr to point to the network address table. Notice that as a result of the ensuing omopen call the list becomes optimized, because net_adr_tbl_ptr is not NULL.

This also sets net_open_list_ptr to point to the variable list, sets the maximum and current sizes of the network address table to 5 and 0, respectively, sets the maximum and current sizes of variables list to 10 and 10, and sets the access to read only. Actually, the sizes can be anything you wish.

OM_R_ACCESS, OM_W_ACCESS, and OM_RW_ACCESS are all defined via the include file.

The following code example is for setting up the variable list:

```
strcpy (variable_list[0].name, name);
variable_list[0].var_desc      = NOTIFY;
variable_list[0].delta         = delta_value;
variable_list[0].var_stat      = 0
```

This example sets up the variable list just for element zero. You would have to set it up for all elements you want to connect to, but not necessarily all of the elements in the list.

The Null-terminated variable name string is copied into the name member. Names should be entered in uppercase. The NOTIFY option is specified in the var_desc member, and the change driven delta value is stored in the delta member. .

The following code opens the list. The list is specified using the address of the header node structure. The ID of the opened list is returned, along with any resulting error code.

```
result = omopen(&om_descriptor, &open_id);
```

No further initialization is needed to reopen a list (optimized or not) after it is closed.

It is essential that, aside from initialization, you do not alter these structures once the list has been opened, except through Object Manager calls. This is especially true for an optimized list, even when closed, if it may be opened again.

3.4.3 omopen Connections

When a list containing optimized remote variables is opened, a connectionless send of a omopen message goes to each remote station believed to have some of the optimized variables. The message consists of one or more packets. A packet contains a copy of the list header as well as the list entries for as many variables as will fit in the packet.

When a station receives an *omopen* packet, the packet is processed by the *server_omopen* routine. This routine allocates, initializes, and opens a new list of the variables specified in the *omopen* packet. If the list has already been opened, the list is extended to contain the variables specified in this latest packet. The list already exists if the packet was not the first packet received for the *omopen* message. The list would also exist if an *open* broadcast message had already been received for the same list.

The variables specified by the *omopen* packet are expected to be local to the station to which the packet is sent. The variables when located, are added to the scanner database, and secured if the list is used for write access.

3.4.4 omopen Server Broadcast

When a list containing unoptimized remote variables is opened, a connectionless broadcast of an *omopen* broadcast type message is issued. The message consists of one or more packets. Each packet contains a copy of the list header as well as the names and some information about as many of the non-local variables will fit in the packet.

When a station receives an omopen broadcast packet, the packet is processed by the *server_omopen_bcast* routine. This routine allocates, initializes, and opens a new list of the variables specified in the broadcast *omopen* packet. If the list has already been opened, the list is extended to contain the variables specified in this latest packet. The list already exists if the packet was not the first packet received for the broadcast message. The list would also exist if an *omopen* message had already been received for the same list. The variables specified by the broadcast packet are searched for locally. The variables, when located, are added to the scanner database, and secured if the list is used for write access.

A remote station knows when it has received all of the *omopen* and *open* broadcast messages for a particular list. The total numbers of optimized and unoptimized variables to expect are contained in every message packet. When enough message packets are received to account for all of these variables, and when these packets have been processed, the remote list is considered opened.

Once the remote copy of the list is complete, the remote station sends an open return type message back to the original station. This message indicates that the remote open has been completed, whether there were variables that were found, and whether the open failed.

No remote list is created if the station does not have any of the variables represented in the message packets. If any *omopen* message packets (representing optimized variables) were received, the *open* return message is sent even if none of the optimized variables could be found. If a station receives only *open* broadcast message packets (representing unoptimized variables), and does not have any of the variables, it will not send an *open* return message.

3.4.5 omopen Server Return

The *server_omopen_return* routine processes the *omopen* return message when it is received by the original station. The corresponding list is found. If the list is optimized, the key code from the remote station is compared against that in the corresponding entry in the Network Address Table entry. If the key code has changed, the optimization change bit is set, and the Network Address Table is updated.

Error status from the remote list is checked and incorporated into the status of the master (local) copy of the list. If there was an error, any relevant change-driven variables are flagged, and the list is posted to the change queue.

Entry information needed from the remote list is copied into the master list. The count of stations which have not yet responded to the *omopen* message is decremented, if appropriate. The count of unoptimized variables which have not yet been accounted for is also decremented, if appropriate.

If the list has become optimized, the link open variables list is called to set up the network address links.

3.5 omread – Read Values From Opened list

omread reads variables from the list opened with the *omopen* call. This call allows you to read all or any part of the list. You can read parts of the list in any order. This call is asynchronous. You must check the status in the value structure to validate the data's integrity.

Format:

```
int omread(<omopen_id>,<size_list>,<var_list>)
    int             omopen_id;
    int             size_list;
    struct value    *var_list;
```

Where:

omopen_id The open_id returned by the omopen call.

size_list The number of value structures in the list.

***var_list** A pointer to the value array structure.

value An array of structures containing the following elements:.

```
struct value {
     int           index;
     unsigned int  status;
     union         {
                   char     letter;
                   int      word;
                   long     longint;
                   float    fpoint;
     } uval;
};
```

index The index value into the variable list corresponding to the variable that changed.

status The data type and status of the variable that changed:

0 = This value not read yet, check for error

Data type – bits 0 to 4:

1 = CHARACTER	5 = BOOL
2 = INTEGER	6 = OM_LNG_INT or POINTER
3 = FLOAT or REAL	9 = OM_PACKED_BOOL
4 = STRING	10 = OM_LONG_PACKED_BOOL

Error status – bits 5 to 7:

0 = No response	4 = Bad data type (string
1 = Being scanned	or unconnectable compound

2 = Disconnected	5-6 = (not used)
3 = Deleted	7 = Not sent (error returned)

`uval` The actual value of the variable.

Return Codes:

`OM_SUCCESS`	The list has been read.
`ESECURE`	Attempt to open a secured variable.
`ENOTOPENED`	List is not opened (or bad om_descriptor).
`EREAD`	The omopen call was set for write only.
`ECONNBAD`	Unable to connect to remote station.
`ESCANFUL`	List open, local scanner database too small; variables that fit in the database are being scanned.
`OM_ECBSY`	Connection was busy on at least one station. The others are opened. There is no auto-reconnect.

Call Notes:

1. For multiple errors, the return code shows only one. EREADERROR is superceded by ESCANFUL, which is superceded by ECONNBAD.
2. To only read part of the list, you can specify the order. If you read the whole list, it reads in ascending order.
3. If one of the status values is zero, that variable has not been read (or perhaps scanned) yet, or there is a problem such as bad data type or variable not found. Check for an error return. In any case, uval does not contain a legitimate value.
4. If you wish to read all variables in the open list, match size_list to the size of the open list. Do not set the individual indices. The Object Manager reads the list in ascending order of index number.
5. There is a macro, v_varlist(s), that statically or dynamically allocates the value structure for you. This macro is defined in the om_user.h include file as:

```
#define v_varlist(s) valloc(sizeof(struct value)*s)
```

The following is an example of how this macro could be used:

```
size_list = 10;
value_list = v_varlist(10);
if (value_list == (struct value *) NULL)
{
        ....
}
```

The first line sets the size_list calling argument. The second line dynamically allocates a 10-element array of "value" structures. If you use this macro, be sure to free the memory when the structure is no longer needed. It is also advisable to check that the macro did not return a NULL pointer indicating insufficient available memory.

3.6 omwrite – Write Values to Opened List

omwrite writes variables to the list opened with the *omopen* call. This call allows writes to any or all entries in a list. *omwrite* is asynchronous if the variable is not in the local station; it initiates the omwrite message and returns control to your process/task.

getval can verify any *omwrite* regardless of the delta. If the write changes the value by the variable list delta, it can be verified by either of two ways:

1. If the variable NOTIFY option is set, use *dqchange* or *dqlist* to verify the write.
2. An *omread* issued after the list scan rate period can also be used.

Format:

```
int omwrite(<omopen_id>,<size_list>,<var_list>);
    int             omopen_id, size_list;
    struct value    *var_list;
```

Where:

`omopen_id`	The open_id returned by the omopen call.
`size_list`	The number of value structures in the var_list.
`*var_list`	A pointer to the struct value array.

A value structure contains the following elements:.

```
struct value {
     int            index;
     unsigned int   status;
     union          {
                    char            letter;
                    int             word;
                    long            longint;
                    unsigned int    uword;
                    unsigned long   ulong;
                    float           fpoint;
     } uval;
} ;
```

Where:

`index`	The number (0 base) of the variable list entry to be written.
`status`	The data type of the variable (low order byte filled in by the caller). An error return code (next to low order byte): 1 = Variable not on scan 2 = The user specified data type does not match the actual data type. 3 = The index is <0 or greater than the size of the list.
`uval`	The value to be written.

Return Codes:

OM_SUCCESS	The list has been written.
ENOSPACE	System resource availability problem.
ENOTOPENED	The specified list has not been opened successfully. No writes are performed.
ESECURE	The specified list has not been opened successfully because a variable in the list has been secured by another open list. No writes are performed.
EWRITE	The list is opened for read-only access. No writes are performed.
OM_ECBSY	The list was not opened successfully because one or more server stations have used up all its allotted IPC channels. No writes are performed.
ECONNBAD	The list was not opened successfully due to one of the following: 1. A remote scanner data base is full. 2. Optimized network address table is full. 3. IPC send problems. No writes are performed.
ESTATION	Not all remote variables in the list have been found/opened. No writes are performed.
EOMWSIZE	The size of the write variable array is < = 0 or greater than the size of the open list. No writes are performed.
ENOTACTIVE	An IPC error was encountered when sending an omwrite message to a remote server station. Variables may have been written to other stations.

For each variable in the write list, validity checks are performed. If an error is encountered, an error code is set in the high byte of the caller's write list variable status. All variables are checked, even if an error condition is encountered in the loop. An EWAIT return has priority over EWRITERROR. If any error is encountered no writes are performed.

EWAIT	A variable is not on scan. The (write list variable status) error code is set to 1.
EWRITEERROR	The write list variable type (set in status) does not match the open list variable type. The error code is set to 2.
EWRITEERROR	The write list variable index is greater than the size of the list (header cur_size_open_list). The error code is set to 3.
EWRITEERROR	A CP write error has occurred. Variables may have been written to other stations. The error code is set to 4.

Call Notes:

1. If you only write part of the list, you can specify the order by index number. If you write it all, (size_list is the same size as the open list) it writes in ascending order regardless of index numbers.
2. If one of the status values is zero, that variable has not been read (or perhaps scanned) yet, or that there is a problem such as bad data type or variable not found. Check for an error return. In any case, uval does not contain a legitimate value.

3. If you wish to write all variables in the open list, match size_list to the size of the open list. Do not set the individual indices. The Object Manager writes the list in ascending order of index number.
4. There is a macro, v_varlist(s), that statically or dynamically allocates the value structure for you. This macro is defined in the om_user.h include file:

```
#define v_varlist(s) valloc(sizeof(struct value)*s)
```

The following is an example of how this macro could be used:

```
size_list = 10;
value_list = v_varlist(10);
if (value_list == (struct value *) NULL)
{
            ....
}
```

The first line sets the size_list calling argument. The second line dynamically allocates a 10-element array of "value" structures. If you use this macro, be sure to free the memory when the structure is no longer needed. It is also advisable to check that the macro did not return a NULL pointer indicating insufficient available memory.

5. Masks are not specified for data types OM_S_PKBOL and OM_L_PKBOL.

3.7 omwrstat – Write Values and/or Status to Opened List

omwrstat writes variable data and/or status to a list opened with the *omopen* call. This call allows writes to any or all entries in a list. *omwrstat* is asynchronous if the variable is not in the local station; it initiates the omwrite message and returns control to your process/task.

getval can verify any *omwrite* regardless of the delta. If the write changes the value by the variable list delta or changes the status, it can be verified by either of two ways:

1. If the variable NOTIFY option is set, use *dqchange* or *dqlist* to verify the write.
2. An *omread* issued after the list scan rate period can also be used.

Format:

```
int omwrstat(<omopen_id>,<size_list>,<var_list>);
    int             omopen_id, size_list;
    struct valstat  *var_list;
```

Where:

`omopen_id`	The open_id returned by the *omopen* call.
`size_list`	The number of valstat structures in the var_list.
`*var_list`	A pointer to the struct valstat array.

A valstat structure contains the following elements:.

```
struct valstat {
     int             index;
     unsigned int    status;
     unsigned int    setstat;
     unsigned int    setmask;
     unsigned int    setonly;
     union           {
                     char            letter;
                     int             word;
                     long            longint;
                     unsigned int    uword;
                     unsigned long   ulong;
                     float           fpoint;
     } uval;
} ;
```

Where:

`index`	The number (0 base) of the variable list entry to be written.
`status`	The data type of the variable (low order byte filled in by the caller). An error return code (next to low order byte). 1 = Variable not on scan.

2 = The user specified data type does not match the actual data type.
3 = The index is <0 or greater than the size of the list.
4 = Setmask contains invalid bits.,

setstat	The bit settings of those bits specified to be written in setmask.
setmask	The status bits to be written or NULL. Valid bits are OM_STAT_BAD and OM_STAT_OOS as defined in OM_UDEF.H.
setonly	If TRUE, write status only for this valstat entry. Ignore uval.
uval	The value to be written.

Return Codes:

OM_SUCCESS	No error.
ENOSPACE	System resource availability problem.
ENOTOPENED	The specified list has no been opened successfully. No writes are performed.
ESECURE	The specified list has not been opened successfully because a variable in the list has been secured by another open list. No writes are performed.
EWRITE	The list is opened for read-only access. No writes are performed.
OM_ECBSY	The list was not opened successfully because one or more server stations have used up all its allotted IPC channels. No writes are performed.
ECONNBAD	The list was not opened successfully due to one of the following: 1. A remote scanner data base is full. 2. Optimized network address table is full. 3. IPC send problems. No writes are performed.
ESTATION	Not all remote variables in the list have been found/opened. No writes are performed.
EOMWSIZE	The size of the write variable array is < = 0 or greater than the size of the open list. No writes are performed.
ENOTACTIVE	An IPC error was encountered when sending an omwrite message to a remote server station. Variables may have been written to other stations.

For each variable in the write list, validity checks are performed. If an error is encountered, an error code is set in the high byte of the caller's write list variable status. All variables are checked, even if an error condition is encountered in the loop. An EWAIT return has priority over EWRITERROR. If any error is encountered no writes are performed.

EWAIT	A variable is not on scan. The (write list variable status) error code is set to 1.
EWRITEERROR	The write list variable type (set in status) does not match the open list variable type. The error code is set to 2.

EWRITEERROR The write list variable index is greater than the size of the list (header cur_size_open_list). The error code is set to 3.

EWRITEERROR The setmask contains bits set other than OM_STAT_OOS or OM_STAT_OOS. The error code is set to 4.

Call Notes:

1. If you only write part of the list, you can specify the order by index number. If you write it all, (size_list is the same size as the open list) it writes in ascending order regardless of index numbers.
2. If one of the status values is zero, that variable has not been read (or perhaps scanned) yet, or that there is a problem such as bad data type or variable not found. Check for an error return. In any case, uval does not contain a legitimate value.
3. If you wish to write all variables in the open list, match size_list to the size of the open list. Do not set the individual indices. The Object Manager writes the list in ascending order of index number.
4. There is a macro, v_varlist(s), that statically or dynamically allocates the value — structure for you. This macro is defined in the om_user.h include file:

   ```
   #define v_varlist(s) valloc(sizeof(struct value)*s)
   ```

   The following is an example of how this macro could be used:

   ```
   size_list = 10;
   value_list = v_varlist(10);
   if (value_list == (struct value *) NULL)
   {
                ....
   }
   ```

   The first line sets the size_list calling argument. The second line dynamically allocates a 10-element array of "value" structures. If you use this macro, be sure to free the memory when the structure is no longer needed. It is also advisable to check that the macro did not return a NULL pointer indicating insufficient available memory.
5. Masks are not specified for data types OM_S_PKBOL and OM_L_PKBOL.

4. C Calls to Locate and Catalog Objects

Locating an object means getting its address. Cataloging means installing or removing objects from the object directory or the import list.

The object directory contains all the shared objects created by the tasks in the local station.

The import list contains the names of all data objects that tasks in the local station want to import from remote stations. Listed with each name is an index value into the address table, which contains the address of the named object.

The OM C calls to locate and catalog objects are listed in this Section in alphabetical order and summarized in Table 4-1.

Table 4-1. C Calls to Locate and Catalog Objects

Paragraph No.	Call	Function
4.1	global_find	Find and object's address
4.2	import	Add object to import list
4.3	obj_create	Add a new object to the directory
4.4	obj_delete	Delete from object directory
4.5	obj_multi_create	Add multiple objects to the directory
4.6	unimport	Remove object from import list

4.1 global_find – Find an Object's Address

The *global_find* call finds an object and returns its address. This call is synchronous; the call suspends until the operation is complete.

Format:

```
int global_find(<obj_name>,<obj_type>,<psap_ptr>)
    char        *obj_name;
    int         obj_type;
    PSAP_ADDR   *psap_ptr;
```

Where:

`*obj_name`	A pointer to the name of the object to be located.
`obj_type`	The object's type, which can be one of the following: VARIABLE, ALIAS, PROCESS, or DEVICE.
`*psap_ptr`	A pointer to the location where `global_find` returns the object's station address.

Return Codes:

`OM_SUCCESS`	The call successfully returned a valid address.
`EBADTYPE`	The specified object type is invalid.
`ENOTFOUND`	The named object could not be found.
`EBADNAME`	The object name is invalid.
`ENOSPACE`	There is a memory allocation error.
`EIPCRET`	An unspecified IPC error.

Call Note:

The address buffer is modified if the call fails. Be sure to check for an error return before using the address.

4.2 import – Add Object to Import List

The *import* call adds a remote object to the local station's import list. The *import* call is synchronous; your task suspends until the call is complete.

Format:

```
int import(<name>,<obj_type>)
    char        *name;
    int         obj_type;
```

Where:

`*name`	A pointer to the name of the object to be imported.
`obj_type`	The object's type, which can be one of the following: VARIABLE, ALIAS, PROCESS, or DEVICE.

Return Codes:

`OM_SUCCESS`	The object is now on the import list.
`EBADTYPE`	The specified object type is invalid.
`EBADNAME`	The specified object name is invalid.
`ENOTFOUND`	The object name cannot be found.
`ENOSPACE`	There is a memory allocation error.
`ELOCAL`	The object you are trying to import is local.
`EIMPFULL`	The import list is too full to add your object.
`EIPCRET`	An unspecified IPC error.

Call Note:

If you move an object of type PROCESS or DEVICE, the import table is not updated. You must unimport, then import these object types to make sure you have the most current address.

4.3 obj_create – Add a New Object to the Directory

The *obj_create* call adds a shared object to the object directory and automatically makes it global. This call is synchronous; your task suspends until the call completes.

Format

```
int obj_create(<name>,<obj_type>,<var_type>,<str_len>)
    char        *name;
    int         obj_type, var_type, str_len;
```

Where:

`*name`	A pointer to the object's name.
`obj_type`	The object's type, which can be one of the following: VARIABLE, ALIAS, PROCESS, or DEVICE.
`var_type`	The data type of the object's value, which can be one of the following: CHARACTER, INTEGER, FLOAT, STRING, OM_LNG_INT, OM_BOOL, OM_S_PKBOL, or OM_L_PKBOL. An alias must be of variable type STRING. `var_type` should be zero for object types PROCESS and DEVICE.
`str_len`	The string length for VARIABLE and ALIAS objects.

Return Codes:

`OM_SUCCESS`	The new object is now in the object directory.
`ESTRLEN`	The string length is invalid.
`EBADTYPE`	Either the object or data type is invalid.
`EBADNAME`	The specified object name is invalid.
`EDUPLICATE`	That object name already exists in the directory.
`EOBJPND`	That object is being created by another task.
`EIPCRET`	An unspecified IPC error.
`ENOSPACE`	No memory available, try again later.

4.4 obj_delete – Delete From Object Directory

The *obj_delete* call removes an object from the object directory. It is asynchronous; it initiates the request then immediately returns control to your task.

Format:

```
int obj_delete(<name>,<obj_type>)
    char        *name;
    int         obj_type;
```

Where:

`*name`	A pointer to the name of the object to be deleted.
`obj_type`	The object's type, which can be one of the following: VARIABLE, ALIAS, PROCESS, or DEVICE.

Return Codes:

`OM_SUCCESS`	The request has been initiated successfully.
`EBADNAME`	The specified object name is invalid.
`EBADTYPE`	The specified object type is invalid.
`ENOTFOUND`	The specified object is not in the directory.

Call Note:

If a deleted object was connected, it becomes disconnected and any task trying to read or write it is notified that it is disconnected (in the status part of the value structure). Since there is no error or warning returned when this happens, it is up to you to make sure you do not delete the wrong objects.

4.5 obj_multi_create – Add Multiple Objects to the Directory

The *obj_multi_create* call adds multiple shared objects to the object directory from a single Object Manager call. All objects are verified for uniqueness, and all objects share a common network timeout period. This call is synchronous; your task suspends until the call completes.

Format

```
int obj_multi_create (<obj_ptr>, <num_objects>)
    struct object_plus    *object_ptr
    int                   num_objs
```

Where:

`obj_ptr:` The object's pointer to the array of `object_plus` object definitions structures (from `om_udat.h`)

`object_ptr->name[ ]:`
NAME_LEN+1 alphanumeric characters of the object's name

`object_ptr->object_type:`
Object type (VARIABLE, ALIAS, PROCESS, DEVICE, etc.)

`object_ptr->type:`
The data type of the object's value, which can be one of the following: CHARACTER, INTEGER, FLOAT, STRING, OM_LNG_INT, OM_BOOL, OM_S_PKBOL, or OM_L_PKBOL.

`object_ptr->str_len:`
If the type is STRING, the string length.

`num_objects:` The number of objects to be created (< or = MAX_REG_SIZE from `om_udef.h`)

Outputs:

`object_ptr->status:`
`object_plus` structure member updated for each object of the array

Return Codes:

`OM_SUCCESS` The new object is now in the object directory

The following returns might be considered warnings, and as such, some of the objects in the array could have been created. In the event more than one of them has occurred, they are returned in the order listed.

`ENOSPACE` The object directory is full

`EOBJPND` The object in the array is a duplicate of an object pending creation

`EDUPLICATE` An object in the array is a duplicate - local or remote

The following errors are considered syntactical errors and prevent all objects from being created.

`EBADTYPE` Either the type is out of range or invalid for the object type

EBADNAME	The specified object name has a non-alphanumeric character, or was too long
ESTRLEN	A STRING variable's str_len was improper
EPORT	Object Manager internal error

4.6 unimport – Remove Object From Import List

The *unimport* call removes an object from the local station's import list. The *unimport* call is asynchronous; it initiates the request then immediately returns control to your task.

Format:

```
int unimport(<name>,<obj_type>)
    char    *name;
    int     obj_type;
```

Where:

`*name`	A pointer to the name of the object to be removed.
`obj_type`	The object's type, which can be one of the following: VARIABLE, ALIAS, PROCESS, or DEVICE.

Return Codes:

`OM_SUCCESS`	The object was successfully removed.
`EBADTYPE`	The specified object type is invalid.
`EBADNAME`	The specified object name is invalid.

5. FORTRAN Functions to Get/Set Object Values

These functions read and write object values. They are connectionless functions, best suited for situations in which you want a single transfer of data. As explained in the document on Inter-Process Communications, connectionless functions are functions that transfer data between tasks without having first established an IPC connection (channel). The Object Manager registers your task with IPC for connectionless service. Be sure to declare the data type of the function name if you are interested in checking possible error returns.

FORTRAN Object Manager programs for the AP10/AP20 I/A Series stations use a 16-bit, integer*2 for object values. The format for each function described below uses the inter*2 for object values.

FORTRAN Object Manager programs for the 50 Series Application programs must adhere to the following conventions:

- All integers must be either INTEGER or INTEGER*4 (not INTEGER*2).
- Access to OM Control and I/O BOOLEAN Variables must be made through the 50 Series FORTRAN compiler type: BYTE.

To compile FORTRAN programs on 50 Series processors, use the command string:

```
f77 tst.f -lfox -lOM77
```

The FORTRAN functions to get/set object values are listed in this Section in alphabetical order and summarized in Table 5-1.

Table 5-1. FORTRAN Functions to Get/Set Object Values .

Paragraph No.	Call	Function
5.1	GETVAL	Get the value of an object
5.2	SETCON	Set the value of an object
5.3	SETVAL	Set the value of an object

5.1 GETVAL – Get the Value of an Object

Use *GETVAL* to read the value of a specified object. This function works for shared variables, process variables, and aliases. *GETVAL* is synchronous; it suspends your task until *GETVAL* completes.

Format:

```
X = GETVAL (<name>,<objtyp>,<import>,<value>,<status>,datlen)
    integer*2          GETVAL
    character*33       name
    integer*2          import, status, datlen, objtyp
    character*x        value
    (or real           value)
    (or integer        value)
    (or integer*2      value)
```

Where:

name	The name of the object.
objtyp	The named object's type; VARIABLE or ALIAS.
import	1 = place the object's name on the import list. 0 = do not place it on the import list.
value	The object's value. Specify its size in bytes using datlen.
status	The object's status (data type). An ALIAS object type can only have character data type. The regular shared VARIABLE object type can have a data type of CHARACTER, INTEGER, INTEGER*2, or REAL. The Object Manager does not restrict process VARIABLE data types, but they can be of types CHARACTER, INTEGER, INTEGER*2, and REAL. Integers are signed, unless the block parameter definition specifies otherwise. Reals adhere to the IEEE standard.
datlen	The length (bytes) allocated for "value."

Return Codes:

OM_SUCCESS	No errors.
ENOTFOUND	Object not found (locally or globally).
EBADTYPE	Illegal object type.
ENOSPACE	No memory available; no value returned.
EBADNAME	Invalid object name.
EBADLEN	Shared variable is too big for datlen.
TO_BIG	Process variable is too big for datlen.
ENOVALUE	Specified object type has no value.

EBADVREC	Value is corrupt, invalid data type.
EIMPFULL	The import list is full; value returned, but not added to list.
EIPCRET	You have an unspecified IPC error.

Call Notes:

1. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects GETVAL ignores that option. However, the value is still returned.
2. Return code ENOTFOUND might mean that the object does not exist locally or globally. If the object exists in a remote station, then that station might be down. In either case, The Object Manager removes the object's name from the import list.
3. If you do not make datlen large enough, value will be too small to hold the returned value. In such a case GETVAL returns as much data in value as will fit.
4. If datlen is too large, value will contain unused bytes. GETVAL does not close-up this unused memory space.

5.2 SETCON – Set the Value of an Object

SETCON sets the value of a specified object and waits for confirmation that it was set. This call works for shared variables, process variables, and aliases. *SETCON* is synchronous; it initiates the request then suspends processing until the operation is complete.

Format:

```
X = SETCON (<name>,<objtyp>,<import>,<value>,<status>,<datlen>)
    integer*2          SETCON
    character          name
    integer*2          import, datlen, status, objtyp
    character*x        value
    (or real           value)
    (or integer        value)
    (or integer*2      value)
```

Where:

name	The name of the object.
objtyp	The named object's type; VARIABLE or ALIAS.
import	1 = Place the object's name on the import list. 0 = Do not place it on the import list.
value	The value to be set. Specify its size in bytes using datlen if it is a string or process variable.
status	The object's data type. The possible data types are listed in FORTRAN Functions to Get/Set Object Values on page 79.
datlen	The length (in bytes) allocated for "value." This is only needed for character data types and all process variables (regardless of data type).

Return Codes:

OM_SUCCESS	The value has been successfully set.
NOT_SETTABLE	This is not a settable process variable.
SECURED	Process variable secured; can't be set.
ENOTFOUND	Object not found (locally or globally).
EBADTYPE	Illegal object type or data type.
ESECURE	Object is secured and cannot be set.
ENOSPACE	No memory available; value not set.
EBADNAME	Invalid object name.
ESTRLEN	The string length is invalid.
ENOVALUE	Specified object type has no "value".
EBADVREC	"Value" is corrupt, invalid data type.
EIMPFULL	The import list is full; value set.

EIPCRET You have an unspecified IPC error.

TO_REAL_CONV_ERROR
Process-variable type-conversion error.

TO_INTEGER_CONV_ERROR
Process-variable type-conversion error.

TO_STRING_CONV_ERROR
Process-variable type-conversion error.

TO_BOOL_CONV_ERROR
Process-variable type-conversion error.

Call Notes:

1. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects SETVAL ignores that option. However, the value is still set.
2. ENOTFOUND might mean that the object does not exist locally or globally. If the object exists in a remote station, then that station might be down. In either case, the Object Manager removes the object's name from the import list.
3. If datlen is too small, SETCON truncates the value to fit.
4. If datlen is too large, value will contain unused bytes. SETCON does not close-up this unused memory space.

5.3 SETVAL – Set the Value of an Object

SETVAL sets the value of the specified object. This function works for shared variables, process variables, and aliases. *SETVAL* is asynchronous; it initiates the request then immediately returns control to your task.

Format:

```
X = SETVAL(<name>,<objtyp>,<import>,<value>,<status>,<datlen>)
    integer*2          SETVAL
    character          name
    integer*2          import, datlen, status, objtyp
    character*x        value
    (or real           value)
    (or integer        value)
    (or integer*2      value)
```

Where:

`name`	The name of the object.
`objtyp`	The named object's type; either VARIABLE or ALIAS.
`import`	1 = Place the object's name on the import list. 0 = Do not place it on the import list.
`value`	The value to be set. Specify its size in bytes using datlen, if it is character or a process variable.
`status`	The object's data type. An ALIAS object type can only have character data type. The regular shared VARIABLE object type can have data types of CHARACTER, INTEGER, INTEGER*2, and REAL. The Object Manager does not restrict process VARIABLE data types, but they can be of types CHARACTER, INTEGER, INTEGER*2, and REAL. Integers are signed, unless the block parameter definition specifies otherwise. Reals adhere to the IEEE standard.
`datlen`	The length (in bytes) allocated for `value`. This is needed only for character data types and all process variables (regardless of data type).

Return Codes:

`OM_SUCCESS`	The set request has been initiated successfully.
`NOT_SETTABLE`	This is not a settable process variable.
`SECURE`	Process variable secured; can't be set.
`ENOTFOUND`	Object not found (locally or globally).
`EBADTYPE`	Illegal object type or data type.
`ESECURE`	Object is secured and cannot be set.
`ENOSPACE`	No memory available; value not set.

EBADNAME	Invalid object name.
ESTRLEN	The string length (data_len) is invalid.
ENOVALUE	Specified object type has no "value".
EBADVREC	"Value" is corrupt, invalid data type.
EIMPFULL	The import list is full; value set.
EIPCRET	You have an Unspecified IPC error.
TO_REAL_CONV_ERROR	Process-variable type-conversion error.
TO_INTEGER_CONV_ERROR	Process-variable type-conversion error.
TO_STRING_CONV_ERROR	Process-variable type-conversion error.
TO_BOOL_CONV_ERROR	Process-variable type-conversion error.

Call Notes:

1. Placing a local object (one found in the same station) on the import list is unnecessary, so for local objects SETVAL ignores that option. However, the value is still set.
2. ENOTFOUND might mean that the object does not exist locally or globally. If the object exists in a remote station, then that station might be down. If so, the Object Manager removes the object's name from the import list.
3. If datlen is too small, SETVAL truncates the value to fit.
4. If datlen is too large, the value will contain unused bytes.

6. FORTRAN Functions to Access/ Update Sets of Variables

These calls allow your task to establish a delta value for each of a list of process variables (using *OMOPEN*). The delta value is how much change has to take place in the variable's value before your task is notified (by *DQCHNG*).

The variables in an opened list are maintained locally so that the response on reads is immediate and requires no network traffic. These calls are designed for tasks needing ongoing access to many process variables.

A task can read from or write to variables on a list opened with *OMOPEN* until the occurrence of until issuing an *OMCLOS*, call, at which time the Object Manager returns a table of addresses where it found the variables. Subsequent *OMOPEN*s can use the same table of addresses, thus saving processing time.

FORTRAN Object Manager programs for the AP10/AP20 I/A Series stations use a 16-bit, integer*2 for object values. The format for each function described below uses the inter*2 for object values.

FORTRAN Object Manager programs for the 50 Series Application programs must adhere to the following conventions:

- All integers must be either INTEGER or INTEGER*4 (not INTEGER*2).
- Access to OM Control and I/O BOOLEAN Variables must be made through the 50 Series FORTRAN compiler type : BYTE.

In addition, when performing reads/writes of OM BOOLEAN types in lists, a four BYTE array must be EQUIVALENCEd to an INTEGER, as shown in the following example:

```
BYTE        BYTEVAL(4)
INTEGER     INTVAL
EQUIVALENCE (BYTEVAL, INTVAL)
OMREAD      (OPENID, NUMVAR, INDEX, STATUS, INTVAL)
```

then to access the byte, reference element 0:

```
VALUE = BYTEVAL(0)
```

The FORTRAN functions to access/update sets of variables are listed in this Section in alphabetical order and summarized in Table 6-1.

Table 6-1. FORTRAN Functions to Access/Update Sets of Variables

Paragraph No.	Call	Function
6.1	DQCHNG	Check for object value changes
6.2	OMCLOS	Close the specified variable list
6.3	OMOPEN	Open a set of variables
6.4	OMREAD	Read values from opened list
6.5	OMWRI	Write values to opened list

6.1 DQCHNG – Check for Object Value Changes

The *DQCHNG* function checks to see if any values in any *OMOPEN*ed variable lists have changed. It checks all the lists for the task named and returns changes in the first variable list it finds. In order to register as a change, the new value has to be greater than or equal to the delta value (in *OMOPEN*) for that value. To be notified, set the NOTIFY option in *OMOPEN*.

Format:

```
X = DQCHNG (<flag>,<openid>,<qsize>,<qindex>,<qstat>,<value>,<rqsize>)
    integer*2       DQCHNG
    integer*2       flag
    integer*2       openid
    integer*2       qsize
    integer*2       qindex(y)
    integer*2       qstat(y)
    integer         value(y)
    integer*2       rqsize
```

Where:

`flag`	0 = don't suspend, 1 = suspend.
`openid`	A value returned to the caller that identifies a previous *omopen* call whose list has changed.
`qsize`	Number of elements in value array.
`qindex(y)`	Where `y = qsize`, the index value into the variable list corresponding to the variable that changed.
`qstat(y)`	An array of status of changed variables.
`value(y)`	An array of values for each entry in the `list`. `value(y)` must be declared in four bytes or floating point, even if the value is character data type.
	Example of a mixed variable type list:
	`rqsize`The number of elements returned.

```
character*4     charva (y)
integer         ival (y)
real            readva (y)
equivalence     (charva, ival, readva)
```

Return Codes:

`OM_SUCCESS`	Changed variables have been successfully returned.
`ESECURE`	One or more variables in the list is secured, so dqchange will not work on this list.
`ENOTOPENED`	The specified task has no variables set for notification access, or no opened lists.

ECONNBAD	The *OMOPEN* could not establish an IPC connection. You should close the list.
EQNOTEMPTY	The value array is too small to hold all the changes that the Object Manager has in its queue.
EQEMPTY	You did not specify the suspend option and there were no changes in the Object Manager's queue.

Call Notes:

1. You can specify any task name (process id) that has open lists. If more than one open list has changes in it, DQCHNG returns the changes in the first list, then remove that list from its queue. Use another DQCHNG call to get the changes in the next list.
2. Each variable list is identified by the open id returned when the list was opened.

6.2 OMCLOS – Close the Specified Variable List

The *OMCLOS* call closes a list of local or remote variables by releasing the connection established with the *OMOPEN* call. It also supplies all of the header information, variable list, and address table to speed up future *OMOPEN* calls to the same list.

OMCLOS is asynchronous; it initiates the request then immediately returns control to your task.

Format

```
X = OMCLOS (<numvar>,<openid>)

    integer*2      OMCLOS
    integer*2      numvar
    integer*2      openid
```

Where:

`numvar`	The number the number of variable names in the specified openid.
`openid`	The id from the previous OMOPEN.

Return Codes:

`OM_SUCCESS`	The request has been initiated.
`ENOTOPENED`	There is no open list corresponding to `open_id`.
`EIPCRET`	You have an unspecified IPC error.

Call Note:

It is imperative that the lists be closed before any exit from the program.

6.3 OMOPEN – Open a Set of Variables

The *OMOPEN* call opens a list of up to 255 local or remote variables. You must use this call before using any of the other data access calls. This call is asynchronous; it initiates the request then immediately returns control to your task.

Format

```
X = OMOPEN (<numvar>,<name>,<delta>,<notify>,<rwacc>,<openid>,<scnrat>)

    integer*2       OMOPEN
    character*33    name(y)
    integer*2       numvar,
                    delta(y),
                    notify(y),
                    rwacc,
                    openid
    integer*2       scnrat
```

Where:

`name`	An array of x names in the list.
`numvar`	The number of variable names.
`delta`	An array of delta values; one for each variable name.
`notify`	An array of change notifications; one for each variable name.
`rwacc`	Read/write access for the list.
`openid`	An id returned to the user.
`scnrat`	The rate at which the station that gets the data sends it to the station that opened the list. This can be a value from 1 to 120 half-second intervals. An invalid scan rate defaults to one (1/2 second interval).

The FORTRAN call only uses the unoptimized version, which takes up less memory. You provide a list of variables, the Object Manager uses its own address table.

Return Codes:

`OM_SUCCESS`	The list has been opened.
`ENOTFOUND`	One or more variables can't be found; the rest are being scanned.
`ESECURE`	Attempt to open a secured variable; the list is open, but no write access to any variables.
`ENOSPACE`	There is not enough room in memory, try again later.
`EOPENED`	The list has already been opened.
`ECONNBAD`	List open but not connected, no write access.
`ENOADDTBL`	Previous *OMOPEN* for this list had an address table; the list is open, but not scanned.
`ENOSEND`	List open, but remotes are not connected or scanned, local variables are being scanned.

EMAXOPNS	The *OMOPEN* id table is full; list not opened.
ESCANFUL	List open, but the local scanner database too small; variables that fit in the database are being scanned.
EIPCRET	You have an unspecified IPC error.

Call Notes:

1. It is faster if the variables you want to access are already listed in the import list. (Refer to the IMPORT call.)
2. All the specified variables are scanned every half-second. You can pick up those with changes greater than or equal to the delta value with a DQCHNG or OMREAD.
3. Unlike the C call, which has both optimized and unoptimized versions, this call is unoptimized. This saves space by using the Object Manager's address table. This table is used by many functions within the local station. Your variable addresses are found and added each time you open the list.
4. Use the openid when you issue the OMCLOS call.
5. It is imperative that the lists be closed before any exit from the program.

6.4 OMREAD – Read Values From Opened list

The *OMREAD* call reads variables from the list opened with the *OMOPEN* call. This call allows you to read all or any part of the list. You can read parts of the list in any order. This call is asynchronous; it returns to your task as soon as it has been initiated. You must check the status in the value structure to see if it completed.

Format

```
X = OMREAD (<openid>,<numvar>,<index>,<status>,<readva>)

    integer*2      OMREAD
    integer*2      openid, numvar, index(y), status(y)
    integer        readva
```

Where:

`openid`	The openid returned by the OMOPEN call.
`numvar`	The number of entries to be read.
`index(y)`	Where x = numvar; the entry you wish to read from the list. If numvar is not specified the default is to read the entire list.
`status(y)`	An array of read statuses for each entry in the list.
`readva(y)`	An array of values for each entry in the list. readva(y) must be declared in four bytes or floating point, even if the value is character data type.

Example of a mixed variable type list:

```
character*4    charva (y)
integer        ival (y)
real           readva (y)
equivalence    (charva, ival, readva)
```

Return Codes:

`OM_SUCCESS`	The list has been read.
`ESECURE`	Attempt to open a secured variable.
`ENOTOPENED`	List is not opened (or bad om_descriptor).
`EREAD`	The *OMOPEN* call was set for write only.
`ECONNBAD`	Unable to connect to remote station.
`ESCANFUL`	List open, local scanner database too small; variables that fit in the database are being scanned.

Call Notes:

1. For multiple errors, the return code shows only one. EREADERROR is superceded by ESCANFUL, which is superceded by ECONNBAD.
2. To only read part of the list, you can specify the order. If you read the whole list, it reads in ascending order.

3. If one of the status values is zero, that variable has not been read (or perhaps scanned) yet, or there is a problem such as bad data type or variable not found. Check for an error return. In any case, uval does not contain a legitimate value.

6.5 OMWRIT – Write Values to Opened List

The *OMWRIT* call writes variables to the array opened with the *OMOPEN* call. This call allows you to write all or any part of the array. You can write parts of the array in any order. This call is asynchronous; it returns to your task as soon as it has been initiated. You must check status to see if it completed.

Format:

```
X = OMWRIT (<openid>,<numvar>,<index>,<status>,<writva>)
```

```
integer*2      OMWRIT
integer*2      openid
integer*2      numvar
integer*2      index(y)
integer*2      status(y)
integer        writva(y)
```

Where:

`openid`	The openid returned by the *OMOPEN* call.
`numvar`	The number of entries to be written.
`index(y)`	Where `x = numvar`; the entry you wish to write to the list. If numvar is not specified the default is to write the entire list in sequential order.
`status(y)`	An array of write statuses for each entry in the list.
`writva(y)`	An array of values for each entry in the list. `writva(y)` must be declared in four bytes or floating point, even if the value is character data type.

Example of a mixed variable type list:

```
character*4 charva (y)
integer ival (y)
real writva (y)
equivalence (charva, ival, writva)
```

Return Codes:

`OM_SUCCESS`	The list has been written.
`ESECURE`	Attempt to write a secured variable.
`ENOTOPENED`	List is not opened (or bad om_descriptor).
`EWRITEERROR`	One or more variables inaccessible (check statuses).
`EWRITE`	The openid indicates a read-only access.
`ECONNBAD`	Unable to connect to remote station.
`EOMSIZE`	Value list is < 1 or larger than opened list.
`EWAIT`	Remote scanner(s) haven't updated the list objects.
`ESTATION`	All remote stations haven't responded.
`ENOSPACE`	Cannot allocate message space for remote write.

EIPCRET You have an unspecified IPC error.

Call Notes:

1. If you only write part of the list, you can specify the order by index number. If you write it all, it writes in ascending order regardless of index numbers.
2. If one of the status values is zero, that variable has not been read (or perhaps scanned) yet, or that there is a problem such as bad data type or variable not found. Check for an error return. In any case, uval does not contain a legitimate value.

7. FORTRAN Functions to Locate and Catalog Objects

Locating an object means getting its address. Cataloging means installing or removing objects from the object directory or the import list

The object directory contains all the shared objects created by the tasks in the local station.

The import list contains the names of all data objects that tasks in the local station want to import from remote stations. Listed with each name is an index value into the address table, which contains the address of the named object.

FORTRAN Object Manager programs for the AP10/AP20 I/A Series stations use a 16-bit, integer*2 for object values. The format for each function described below uses the inter*2 for object values.

FORTRAN Object Manager programs for the 50 Series Application programs must adhere to the following conventions:

- All integers must be either INTEGER or INTEGER*4 (not INTEGER*2).
- Access to OM Control and I/O BOOLEAN Variables must be made through the 50 Series FORTRAN compiler type : BYTE.

The FORTRAN functions to locate and catalog objects are listed in this Section in alphabetical order and summarized in Table 7-1.

Table 7-1. FORTRAN Functions to Locate and Catalog Objects

Paragraph No.	Call	Function
7.1	IMPORT	Add object to import list
7.2	OCREAT	Add a new object to the directory
7.3	ODELET	Delete from object directory
7.4	UNIMPO	Remove object from import list

7.1 IMPORT – Add Object to Import List

The *IMPORT* call adds a remote object to the local station's import list. *IMPORT* is synchronous; your task suspends until the function is complete.

Format

```
int IMPORT(<name>,<objtyp>)
    character*33      name
    integer*2         import, objtyp
```

Where:

*name	A pointer to the name of the object to be imported.
obj_type	The object's type, which can be one of the following: VARIABLE, ALIAS, PROCESS, or DEVICE.

Return Codes:

OM_SUCCESS	The object is now on the import list.
EBADTYPE	The specified object type is invalid.
EBADNAME	The specified object name is invalid.
ENOTFOUND	The object name cannot be found.
ENOSPACE	There is a memory allocation error.
ELOCAL	The object you are trying to import is local.
EIMPFULL	The import list is too full to add your object.
EIPCRET	An unspecified IPC error.

7.2 OCREAT – Add a New Object to the Directory

The *OCREAT* function adds a shared object to the object directory and automatically makes it global. This function is synchronous; your task suspends until the function completes.

Format:

```
X = OCREAT (<name>,<objtyp>,<vartyp>,<datlen>)

    integer*2          OCREAT
    integer*2          objtyp, vartyp, datlen
    character*33       name
```

Where:

`name`	The object's name.
`objtyp`	The object's type, which can be one of the following: VARIABLE, ALIAS, PROCESS, or DEVICE.
`vartyp`	The data type of the object's value, which can be one of the following: CHARACTER, INTEGER, FLOAT, or CHARACTER. An alias must be of variable type CHARACTER. vartyp should be null for object types PROCESS and DEVICE.
`datlen`	The character string length for VARIABLE and ALIAS objects.

Return Codes:

`OM_SUCCESS`	The new object is now in the object directory.
`ESTRLEN`	The string length is invalid.
`EBADTYPE`	Either the object or data type is invalid.
`EBADNAME`	The specified object name is invalid.
`EDUPLICATE`	That object name already exists in the directory.
`EOBJPND`	That object is being created by another task.
`EIPCRET`	An unspecified IPC error.

7.3 ODELET – Delete From Object Directory

The *ODELET* call removes an object from the object directory. It is asynchronous; it initiates the request then immediately returns control to your task.

Format:

```
X = ODELET (<name>,<objtyp>)

    integer*2          ODELET
    character*33       name
    integer*2          objtyp
```

Where:

name	The name of the object to be deleted.
objtyp	The object's type, which can be one of the following: VARIABLE, ALIAS, PROCESS, or DEVICE.

Return Codes:

OM_SUCCESS	The request has been initiated successfully.
EBADNAME	The specified object name is invalid.
EBADTYPE	The specified object type is invalid.
ENOTFOUND	The specified object is not in the directory.

Call Note:

If a deleted object was connected, it becomes disconnected and any task trying to read or write it is notified that it is disconnected (in the status part of the value structure). Since there is no error or warning returned when this happens, it is up to you to make sure you do not delete the wrong objects.

7.4 UNIMPO – Remove Object From Import List

The *UNIMPO* function removes an object from the local station's import list. This function is asynchronous; it initiates the request then immediately returns control to your task.

Format:

```
X = UNIMPO (<name>,<objtyp>)
    integer*2          UNIMPO
    character*33       name
    integer*2          objtyp
```

Where:

`name`	The name of the object to be removed.
`objtyp`	The object's type, which can be one of the following: VARIABLE, ALIAS, PROCESS, or DEVICE.

Return Codes:

`OM_SUCCESS`	The object was successfully removed.
`EBADTYPE`	The specified object type is invalid.
`EBADNAME`	The specified object name is invalid.

Appendix A. OM Calls Sample Programs

A.1 OM Program Creates and Initializes Variables

```
#include <stdio.h>
#include <fox/om_user.h>
#include <fox/om_ecode.h>

main()
{
float value = 0.0;
unsigned int status = FLOAT;
int static data_len;

data_len = 4;
rtn = obj_create("CMP1_TI1_PNT",VARIABLE,FLOAT);
rtn = setval("CMP1_TI1_PNT",VARIABLE,0,&value,&status,&data_len);
printf("RTN = %d\n",rtn);
rtn = obj_create("CMP1_TI2_PNT",VARIABLE,FLOAT);
rtn = setval("CMP1_TI2_PNT",VARIABLE,0,&value,&status,&data_len);
printf("RTN = %d\n",rtn);
rtn = obj_create("CMP1_TI3_PNT",VARIABLE,FLOAT);
rtn = setval("CMP1_TI3_PNT",VARIABLE,0,&value,&status,&data_len);
printf("RTN = %d\n",rtn);
rtn = obj_create("CMP1_TI4_PNT",VARIABLE,FLOAT);
rtn = setval("CMP1_TI4_PNT",VARIABLE,0,&value,&status,&data_len);
printf("RTN = %d\n",rtn);
rtn = obj_create("CMP2_FC1_MEAS",VARIABLE,FLOAT);
rtn =setval("CMP2_FC1_MEAS",VARIABLE,0,&value,&status,&data_len);
printf("RTN = %d\n",rtn);
rtn = obj_create("CMP2_FC2_MEAS",VARIABLE,FLOAT);
rtn =setval("CMP2_FC2_MEAS",VARIABLE,0,&value,&status,&data_len);
printf("RTN = %d\n",rtn);
rtn = obj_create("CMP2_FC3_MEAS",VARIABLE,FLOAT);
rtn =setval("CMP2_FC3_MEAS",VARIABLE,0,&value,&status,&data_len);
printf("RTN = %d\n",rtn);
rtn = obj_create("CMP2_FC4_MEAS",VARIABLE,FLOAT);
rtn =setval("CMP2_FC4_MEAS",VARIABLE,0,&value,&status,&data_len);
printf("RTN = %d\n",rtn);
rtn = obj_create("CMP3_FZ01_RIN1",VARIABLE,FLOAT);
```

```
rtn=setval("CMP3_FZ01_RIN1",VARIABLE,0,&value,&status,&data_len);
printf("RTN = %d\n",rtn);
rtn = obj_create("CMP3_FZ01_RIN2",VARIABLE,FLOAT);
rtn=setval("CMP3_FZ01_RIN2",VARIABLE,0,&value,&status,&data_len);
printf("RTN = %d\n",rtn);
rtn = obj_create("CMP3_FZ01_RIN3",VARIABLE,FLOAT);
rtn=setval("CMP3_FZ01_RIN3",VARIABLE,0,&value,&status,&data_len);
printf("RTN = %d\n",rtn);
rtn = obj_create("CMP3_FZ01_RIN4",VARIABLE,FLOAT);
rtn=setval("CMP3_FZ01_RIN4",VARIABLE,0,&value,&status,&data_len);
printf("RTN = %d\n",rtn);
}
```

A.2 OM Program Opens Two Lists for Reads and Writes

```
#include <stdio.h>
#include <fox/om_user.h>
#include <fox/om_ecode.h>

main()
{
      struct open_var         in_var_list[8], out_var_list[4];
      struct header_node      in_om_desc,     out_om_desc
      struct net_adr          in_net_adr_tbl[2], out_net_adr_tbl;
      int     in_open_id, out_open_id;
      int rtn;
      float delta_temp, delta_fc, delta_df;
      struct value    *in_data_list, *out_data_list, *temp;
      int i;

      delta_temp = 5.0;
      delta_fc = 1.0;
      delta_df = 0.5;

      in_om_desc.task_status = OM_R_ACCESS;
      in_om_desc.net_adr_tbl_ptr = in_net_adr_tbl;
      in_om_desc.size_net_adr_tbl = 2;
      in_om_desc.open_list_ptr = in_var_list;
      in_om_desc.size_open_list = 8;

      out_om_desc.task_status = OM_W_ACCESS;
      out_om_desc.net_adr_tbl_ptr= &out_net_adr_tbl;
```

```
out_om_desc.size_net_adr_tbl = 1;
out_om_desc.open_list_ptr = out_var_list;
out_om_desc.size_open_list = 4;

strcpy(in_var_list[0].name,"CMP1_TI1_PNT");
in_var_list[0].var_desc = NOTIFY;
in_var_list[0].delta = delta_temp;

strcpy(in_var_list[1].name,"CMP1_TI2_PNT");
in_var_list[1].var_desc = NOTIFY;
in_var_list[1].delta = delta_temp;
strcpy(in_var_list[2].name,"CMP1_TI3_PNT");
in_var_list[2].var_desc = NOTIFY;
in_var_list[2].delta = delta_temp;
strcpy(in_var_list[3].name,"CMP1_TI4_PNT");
in_var_list[3].var_desc = NOTIFY;
in_var_list[3].delta = delta_temp;
strcpy(in_var_list[4].name,"CMP2_FC1_MEAS");
in_var_list[4].var_desc = NOTIFY;
in_var_list[4].delta = delta_fc;
strcpy(in_var_list[5].name,"CMP2_FC2_MEAS");
in_var_list[5].var_desc = NOTIFY;
in_var_list[5].delta = delta_fc;
strcpy(in_var_list[6].name,"CMP2_FC3_MEAS");
in_var_list[6].var_desc = NOTIFY;
in_var_list[6].delta = delta_fc;
strcpy(in_var_list[7].name,"CMP2_FC4_MEAS");
in_var_list[7].var_desc = NOTIFY;
in_var_list[7].delta = delta_fc;

strcpy(out_var_list[0].name,"CMP3_FZ01_RIN1")
out_var_list[0].var_desc = NOTIFY;
out_var_list[0].delta = delta_df;
strcpy(out_var_list[1].name,"CMP3_FZ01_RIN2")
out_var_list[1].var_desc = NOTIFY;
out_var_list[1].delta = delta_df;
strcpy(out_var_list[2].name,"CMP3_FZ01_RIN3")
out_var_list[2].var_desc = NOTIFY;
out_var_list[2].delta = delta_df;
strcpy(out_var_list[3].name,"CMP3_FZ01_RIN4")
out_var_list[3].var_desc = NOTIFY;
out_var_list[3].delta = delta_df;

rtn = omopen(&in_om_desc,&in_open_id);
printf("Return = %d\n",rtn);
```

```
rtn = omopen(&out_om_desc,&out_open_id);
printf("Return = %d\n",rtn);

if ((in_data_list = (struct value *)v_varlist (8)) == NULL)
    {
        printf("Can't allocate space \n");
        omclose(in_open_id,&in_om_desc,in_var_list,
          in_net_adr_tbl);
        omclose(out_open_id,&out_om_desc,out_var_list,
          &out_net_adr_tbl);
        exit(0);
    }
sleep(1);
if((rtn = omread(in_open_id,8,in_data_list)) != OM_SUCCESS)
    {
        printf("Return = %d\n",rtn");
        omclose(in_open_id,&in_om_desc,in_var_list,
          in_net_adr_tbl);
        omclose(out_open_id,&out_om_desc,out_var_list,
          &out_net_adr_tbl);
        exit(0)
    }
for (i = 0; i<8; i++)
{
printf("Variable %d = %f\n",i,in_data_list->uval.fpoint);
in_data_list++;
}
if((out_data_list = (struct value *)v_varlist (4)) == NULL)

   {
        printf("Can't allocate space \n");
        omclose(in_open_id,&in_om_desc,in_var_list,
          in_net_adr_tbl);
        omclose(out_open_id,&out_om_desc,out_var_list,
          &out_net_adr_tbl);
        exit(0);
   }
temp = out_data_list;

for (i = 0; i < 4; i++)
{
temp->index = i;
temp->status = FLOAT;
temp->uval.fpoint = 5.0;
```

```
        temp++;
        }

        if((rtn = omwrite(out_open_id,4,out_data_list))
          != OM_SUCCESS)
            {
                printf("Return = %d\n"rtn);

                for(i=0;i<4;i++)
                        {
                        printf("status = %u\n",out_data_list->
                          status);
                        out_data_list++;
                        }
                omclose(in_open_id,&in_om_desc,in_var_list,
                  in_net_adr_tbl);
                omclose(out_open_id,&out_om_desc,out_var_list,
                  &out_net_adr_tbl);
                exit(0);
            }

omclose(in_open_id,&in_om_desc,in_var_list,in_net_adr_tbl);
omclose(out_open_id,&out_om_desc,out_var_list,&out_net_adr_tbl);
```

Appendix B. OM Error Codes

Code	Message	Code	Message
0	Call executed successfully	-30	Dqchange queue is empty
-1	Not found	-31	No address table
-2	Bad data type	-32	Unable to get a queue for notification
-3	Variable is secured	-33	Unable to send open requests
-4	List not opened	-34	Address table full
-5	Error reading list	-35	Not activated with IPC
-6	List write only	-36	Error creating connection table
-7	List read only	-37	Error creating disconnect table
-8	Error writing list	-38	Error creating open id table
-9	Memory allocation error	-39	No space to perform again
-10	Variable already exists	-40	Scanner db full
-11	Variable is local	-41	Error returned from IPC
-12	Name is invalid	-42	Remote scanner has not responded
-13	Warning all data not returned	-43	Remote station has not responded
-14	String length is invalid	-44	Another object of this name is pending creation
-15	List already opened	-45	No confirmation
-16	Bad variable list pointer	-46	Error creating queue for OM VENIX task
-17	Bad address table pointer	-47	Error creating OM scanner task
-18	List has connection problems	-48	Error creating OM VENIX task
-19	Dqchange queue not empty	-49	Error creating OM server task
-20	Object type has no value record	-50	Received fabort
-21	Value record corrupted	-51	Invalid list size
-22	Error creating import list	-52	Error creating reconnect queue
-23	Error creating object directory	-53	Invalid users address table size
-24	Error creating address table	-54	Error creating OM reconnect task
-25	Error creating scanner db		
-26	Import list full		
-27	Error creating dqchange queue table		
-28	Error creating a queue		
-29	Invalid list size		

Appendix C. OM Calls

C.1 OM C Calls Summary

dqchange	int dqchange(<pid>,<suspend>,<open_id>, <size_list>,<value_list>,<ret_size>)
dqlist	int dqlist(<pid>,<suspend>,<open_id>,<size_list>, <value_list>,<ret_size>)
getval	int getval(<name>,<obj_type>,<import>,<value>, <status>,<data_len>)
getval_list	int getval_list(<name>,<obj_type>,<import>, <value>,<status>,<data_len>,<open_id>,<ov_index>)
global_find	int global_find(<obj_name>,<obj_type>,<psap_ptr>)
import	int import(<name>,<obj_type>)
obj_create	int obj_create(<name>,<obj_type>,<var_type>, <str_len>)
obj_delete	int obj_delete(<name>,<obj_type>)
obj_multi_create	int obj_multi_create (<obj_ptr>,<num_objects>)
omclose	int omclose(<open_id>,<header>,<var_list>, <addr_tbl>)
omopen	int omopen(<om_descriptor>,<open_id>)
omread	int omread(<omopen_id>,<size_list>,<var_list>)
omwrite	int omwrite(<omopen_id>,<size_list>,<var_list>)
omwrstat	int omwrstat(<omopen_id>,<size_list>,<var_list>)
om_getval	int om_getval(<name>,<obj_type>,<import>,<value>, <status>,<data_len>,<psap_ptr>)
om_setval	int om_setval (<name>,<obj_type>,<import>,<value>, <status>,<data_len>,<psap_ptr>)
om_set_confirm	int om_set_confirm(<name>,<obj_type>,<import>, <value>,<status>,<data_len>, <psap_ptr>)
set_cnf_list	int set_cnf_list(<name>,<obj_type>,<import>, <value>,<status>,<data_len>,<open_id>, <ov_index>)
set_confirm	int set_confirm(<name>,<obj_type>,<import>, <value>,<status>,<data_len>)
setval	int setval(<name>,<obj_type>,<import>,<value>, <status>,<data_len>)
setval_list	int setval_list(<name>,<obj_type>,<import>, <value>,<status>,<data_len>,<open_id>, <ov_index>)
st_omset_confirm	int st_omset_confirm(<name>,<obj_type>,<import>, <value>,<staus>,<data_len>,<psap_ptr>,<st_data>, <st_mask>,<st_only>,<appl_work>)

st_om_setval	int st_om_setval(<name>,<obj_type>,<import>, <value>,<staus>,<data_len>,<psap_ptr>,<st_data>, <st_mask>,<st_only>,<appl_work>)
st_setcnf	int st_setcnf(<name>,<obj_type>,<import>,<value>, <status>,<data_len>,<st_data>,<st_mask>,<st_only>, <appl_work>)
st_setlist_confirm	int st_setlist_confirm(<name>,<obj_type>,<import>, <value>,<status>, <data_len>,<open_id>,<ov_index>, <st_data>,<st_mask>,<st_only>, <appl_work>)
st_set_list	int st_set_list(<name>,<obj_type>,<import>, <value>,<status>,<data_len>,<open_id>,<ov_index>, <st_data>,<st_mask>,<st_only>,<appl_work>
st_setval	int st_setval(<name>,<obj_type>,<import>,<value>, <status>,<data_len>,<st_data>,<st_mask>,<st_only>, <appl_work>)
unimport	int unimport(<name>,<obj_type>)

C.2 OM FORTRAN Calls

DQCHNG	X = DQCHNG (<flag>,<openid>,<qsize>,<qindex>, <qstat>,<value>,<rqsize>)
GETVAL	X = GETVAL (<name>,<objtyp>,<import>,<value>, <status>,datlen)
IMPORT	int IMPORT(<name>,<objtyp>)
OCREAT	X = OCREAT (<name>,<objtyp>,<vartyp>,<datlen>)
ODELET	X = ODELET (<name>,<objtyp>)
OMCLOS	X = OMCLOS (<numvar>,<openid>)
OMOPEN	X = OMOPEN (<numvar>,<name>,<delta>,<notify>, <rwacc>,<openid>,<scnrat>)
OMREAD	X = OMREAD (<openid>,<numvar>,<index>,<status>, <readva>)
OMWRIT	X = OMWRIT (<openid>,<numvar>,<index>,<status>, <writva>)
SETCON	X = SETCON (<name>,<objtyp>,<import>,<value>, <status>,<datlen>)
SETVAL	X = SETVAL(<name>,<objtyp>,<import>,<value>, <status>,<datlen>)
UNIMPO	X = UNIMPO (<name>,<objtyp>)

Index

EV324849404US

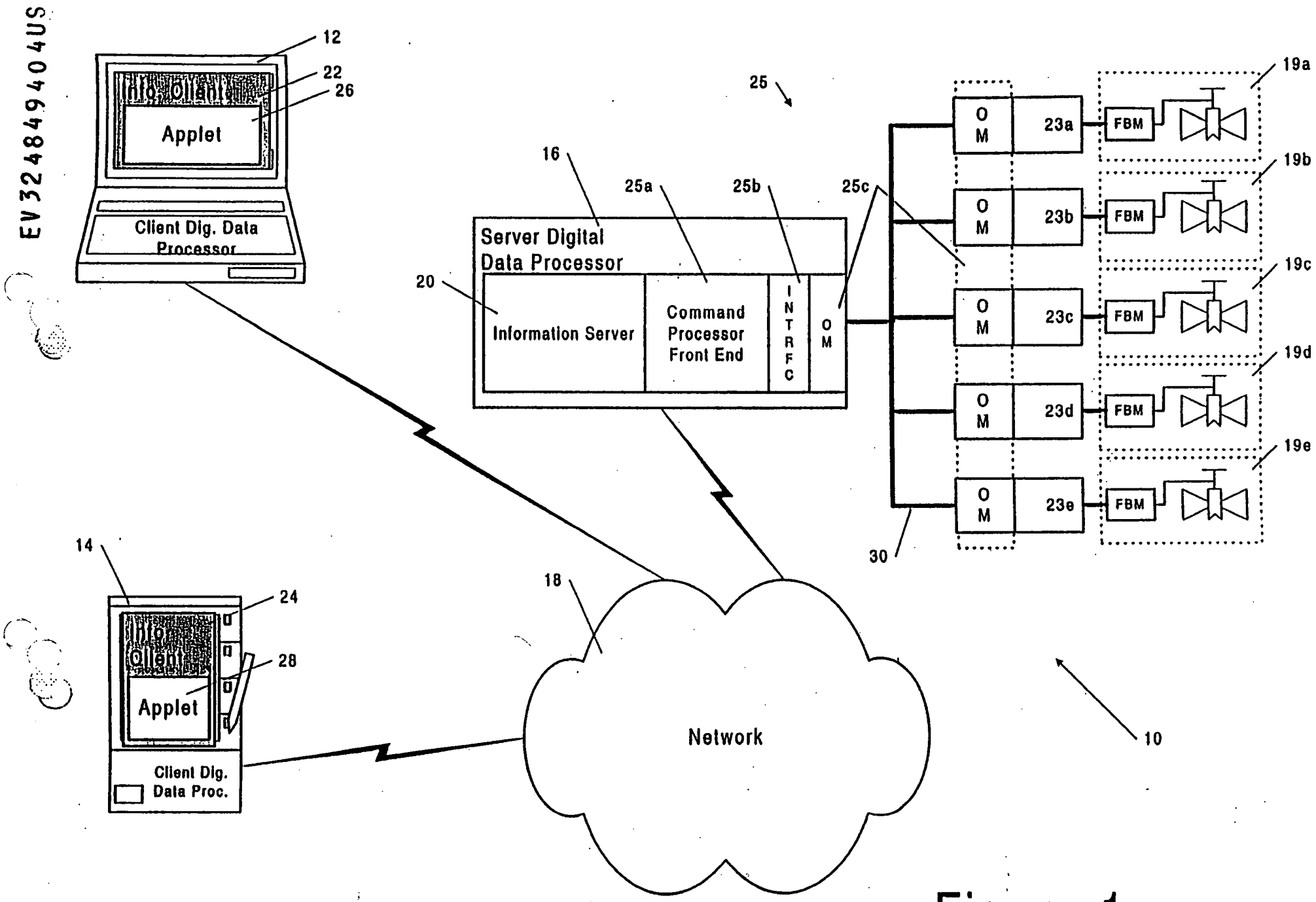


Figure 1

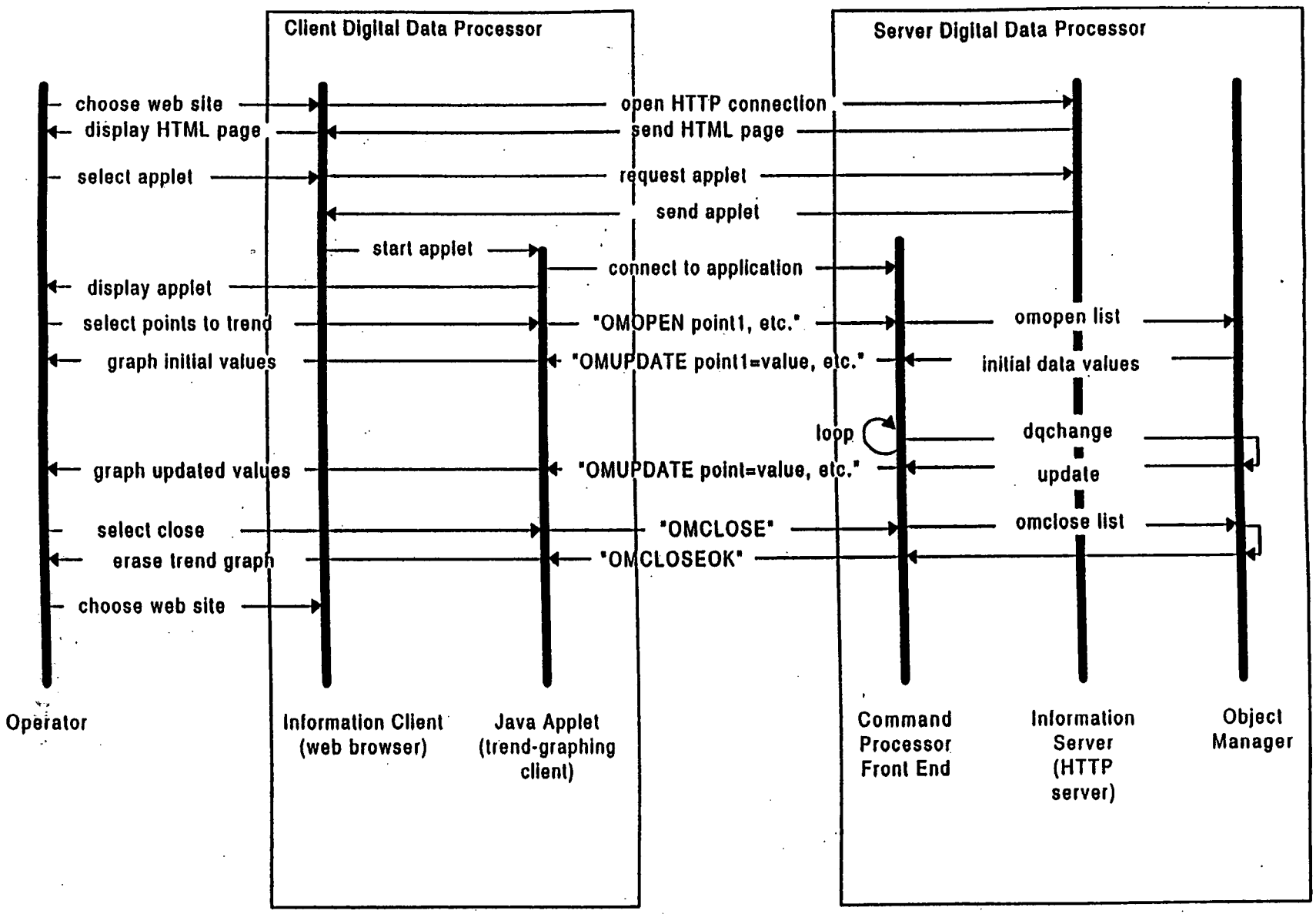


Figure 2